Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Karsten Dieckmann

Schnelle Spektralanalyse auf einem DSP-Bord
mittels optimiertem FFT-Code

Fakultät Technik und Informatik
Department Informations- und
Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and
Electrical Engineering

Karsten Dieckmann

Schnelle Spektralanalyse auf einem DSP- Bord mittels optimiertem FFT-Code

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

Betreuender Prüfer : Prof. Dr. -Ing. Hans Peter Kölzer
Zweitgutachter : Prof. Dr. -Ing. Ulrich Sauvagerd

Abgegeben am 12. Februar 2010

Karsten Dieckmann

Thema der Bachelorthesis

Schnelle Spektralanalyse auf einem DSP- Bord mittels optimiertem FFT-Code

Stichworte

Diskrete Fourier-Transformation, FFT-Algorithmen, DSP-Bord, Codeoptimierung, C-Code, Assembler-Code

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Analyse unterschiedlicher Verfahren zur Berechnung einer diskreten Fourier-Transformation. Diese Verfahren werden auf einem digitalen Signalprozessor „TMS320C6713“ der Firma *Texas Instruments* implementiert und hinsichtlich ihrer Geschwindigkeit und Genauigkeit verglichen.
Als Ergebnis wird ein für den Prozessor optimierter Code zur Berechnung der diskreten Fourier-Transformation entwickelt.

Karsten Dieckmann

Title of the paper

Fast spectral analysis on a DSP-Board using optimized FFT-Code

Keywords

Discrete Fourier transform, FFT-Algorithm, DSP Board, Code improvement, C-Code, Assembler-Code

Abstract

The Bachelorthesis is about the analysis of different methods to calculate a discrete Fourier transform. These methods get implemented on a digital Signal Processor “TMS320C6713” from *Texas Instruments* and are being compared in terms of speed and accuracy.
The result will be an optimised C-Code for this processor to calculate the discrete Fourier transform.

I. Verzeichnisse

I.I. Inhaltsverzeichnis

I.II. Abbildungsverzeichnis

I.III. Tabellenverzeichnis

I.IV. Fremdwortverzeichnis

Branch	Sprung im Programmcode
Cast	Typumwandlung
Central Processing Unit	Hauptprozessor
Cross-Path-Operation	Zugriff auf gegenüberliegende Datenpfade
Cycle	Taktzyklus
Debugging	Fehlersuche
Direct Memory Access	Speicherdirektzugriff
Floating-Point	Fließkommazahl/Gleitkommazahl
Fixpoint	Festkommazahl
Light-Emitting Diode	Leuchtdiode
Optimizer	Funktionseinheit zur Optimierung des Programmcodes
Overhead	Mehraufwand, der nicht direkt Nutzen erzeugt
Pipelining	Fließbandartige Abarbeitung von Programmabschnitten
Profiling	Analyse des Laufzeitverhaltens von Software
Programm Counter	Programmzähler
Random Access Memory	Speicher mit direktem Zugriff
Signal to Noise Ratio	Signal-Geräusch-Abstand
Stack	Stapelspeicher (häufig verwende Datenstruktur bei Mikroprozessoren)

I.V. Liste der Abkürzungen

CPU	Central Processing Unit
DMA	Direct Memory Access
DFT	Diskrete Fourier-Transformation
DSK	DSP Starter-Kit
DSP	Digitaler Signal-Prozessor
EDMA	Enhanced Direct Memory Access
FFT	Fast Fourier-Transformation
JTAG	Joint Test Action Group (Programmier- und Debugschnittstelle)
LED	Light-Emitting Diode
MatLab	Matrix Laboratory
RAM	Random Access Memory
SNR	Signal to Noise Ratio

1 Einführung

Diese Arbeit befasst sich mit der Entwicklung einer hardwarespezifischen Lösung der schnellen Fourier-Transformation (FFT). Dazu werden unterschiedliche Möglichkeiten auf der mathematischen sowie auf der Implementierungsebene untersucht. Das Ergebnis soll ein effizienter Programmcode zur Berechnung der FFT sein.

1.1 Motivation

Im Labor „Digitale Signalverarbeitung“ wird im Rahmen von Diplom-/Bachelor- und Masterarbeiten, sowie im Praktikum, das Prozessorbord „TMS320C6713“ DSK eingesetzt.
Die Berechnung von Spektralanalysen mittels einer DFT/FFT zählt zu den häufig genutzten Anwendungen dieses Bords. Eine schnelle Implementierung der FFT auf diesem Bord würde zu Zeiteinsparungen führen, sodass die gewonnene Zeit für andere Anwendungen genutzt werden könnte. Im Hinblick auf eine Echtzeitverarbeitung der Daten, erscheint es daher sinnvoll, eine effiziente FFT nutzen zu können. Zwar sind bereits diverse Programme zur schnellen Berechnung der FFT in verschiedenen Programmiersprachen und Varianten öffentlich zugänglich, jedoch ermöglichen diese noch keine optimale Ausnutzung des Prozessors, denn dafür ist es notwendig, den entsprechenden Code an den jeweiligen Prozessor anzupassen.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist es, einen hardwarespezifischen Programmcode, zur Berechnung einer FFT, zu entwickeln und diesen bezüglich der Echtzeitfähigkeit zu optimieren. Dazu soll der Code an die gegebene Hardware-Plattform der Firma *Spectrum Digital* mit dem Signalprozessor „TMSC6713“ der Firma *Texas Instruments* angepasst werden. Die zugehörige Entwicklungsumgebung „Code Composer Studio“ soll dabei zur Programmierung genutzt werden.

Folgende Schwerpunkte soll die Arbeit umfassen:

- Untersuchung verschiedener FFT Algorithmen hinsichtlich ihrer Geschwindigkeit, und praktischen Anwendbarkeit
- Implementierung verschiedener Algorithmen auf dem Signalprozessor
- Optimierung der FFT auf der gegebenen Hardware
  - C-Code
  - Assembler-Code

- Geschwindigkeitsmessung
- Bewertung der Ergebnisse

1.3 Konzeptentwurf

Zu Beginn der Arbeit sollen unterschiedliche Lösungsansätze mathematisch betrachtet werden. Dabei soll überprüft werden, welche dieser Algorithmen am besten für eine effiziente Implementierung auf der gegebenen Hardware geeignet sind. Im Anschluss daran sollen die ausgewählten Algorithmen programmiert werden. Dabei soll nicht die Programmierung, sondern die Optimierung des Programmablaufs zur effizienten Ausnutzung der verwendeten Hardware, im Vordergrund stehen. Dazu soll neben der Implementierung der Algorithmen auch ein effizientes Einlesen (Sampeln) und eine notwendige Nachbereitung der Daten verbessert werden.
Zur Umsetzung der Algorithmen sollen diese zunächst in der Programmiersprache C geschrieben werden. Durch eine Analyse sollen rechenaufwändige Programmabschnitte ermittelt werden, um die Laufzeit dieser durch die Nutzung von Assembler Befehlen weiter zu verringern.

2 Hardware- und Software- Komponenten

In diesem Kapitel werden die Hard- und Softwarekomponenten, die zur Erstellung dieser Arbeit zur Verfügung stehen, beschrieben. Bei dem System handelt es sich um das Standardsystem, welches im Labor „Digitale Signalverarbeitung“ zum Einsatz kommt.

2.1 Digitales Signalprozessor-Bord

Das digitale Signalprozessor-Bord „TMS320C6713 DSK“ ist ein komplettes System zur digitalen Signalverarbeitung mit den nötigen Hardwarekomponenten der Firma *Spectreum Digital Inc*. Auf dem Bord befindet sich ein „TMS320C6713 DSP“ Fließkomma- (floting-point) Signalprozessor der Firma *Texas Instruments* und ein 32 Bit Stereo Audio- Codec „TLV320AIC23“ für die Ein- und Ausgabe der Daten. Der Datenaustausch zwischen dem DSP und dem „AIC23“ Codec erfolgt über zwei „Multichannel bufferd serial Ports“ (McBSPs).

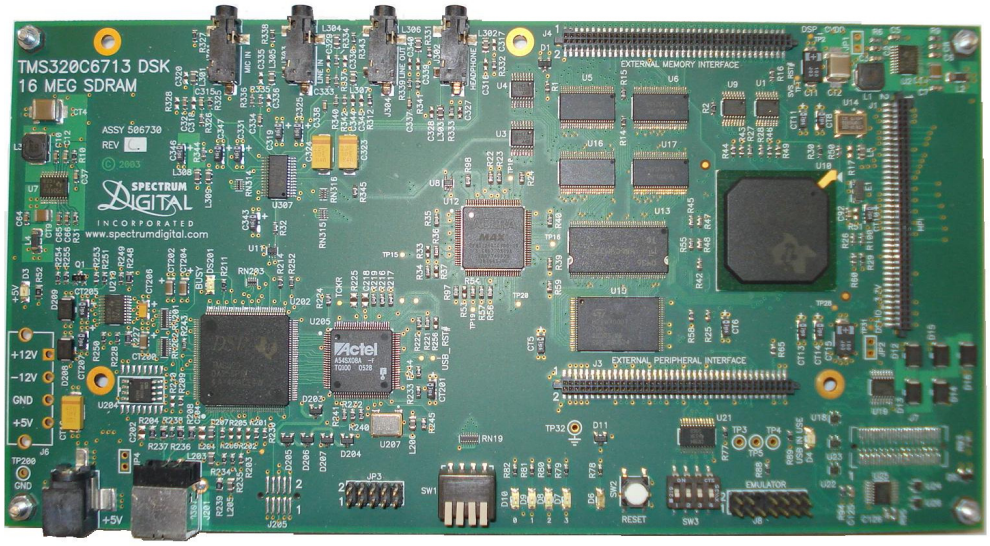


Abbildung 2.1 TMS320 C6713 DSK

Auf dem Bord befinden sich weiterhin ein 16 MB synchroner dynamischer RAM (SDRAM) sowie ein 256 kB Flash-Speicher und ein „Complex Programmable Logic Device“ (CPLD). Als Anschüsse für externe Signale stehen auf dem Bord vier 3,5 mm Klinkenstecker zur Verfügung. Zum einen „MIC IN“ für den Mikrofoneingang sowie „HEADPHONE“ für einen Kopfhörerausgang. Außerdem sind „LINE IN“ und „LINE OUT“ für die Ein- und Ausgabe vorgesehen [27]. Als weitere Peripherie befinden sich vier Dipschalter und vier frei nutzbare LEDs auf dem Bord (s. Abbildung 2.2 [27]).

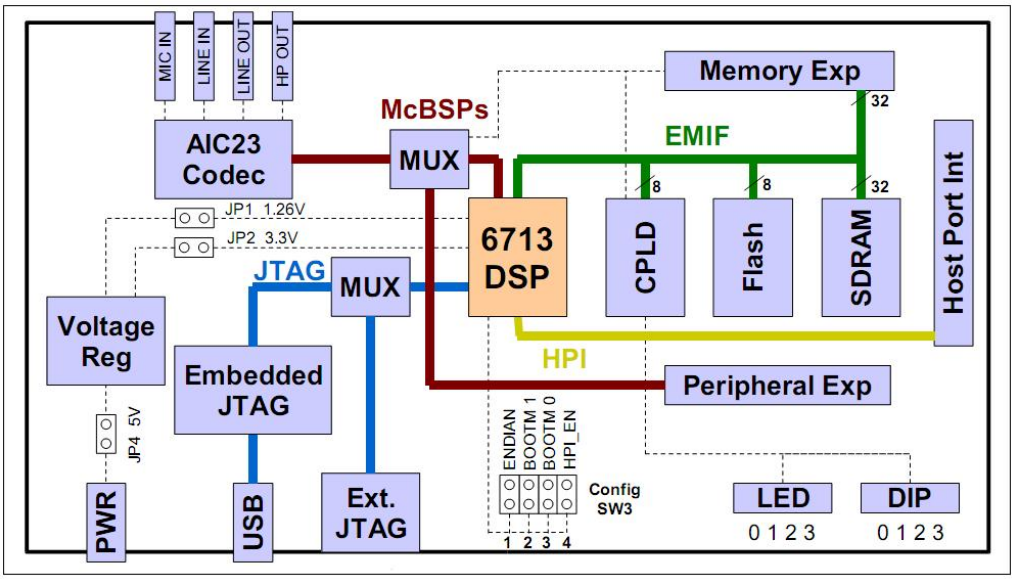


Abbildung 2.2 Block Diagramm C6713 DSK

Programmieren und während der Laufzeit „debuggen“ lässt sich das Bord über eine JTAG (Joint Test Action Group) Schnittstelle. Der Signalprozessor arbeitet auf dem Bord mit einem Takt von 225 MHz und wird über einen Spannungsregler mit 1,26 V versorgt. Die angeschlossene Peripherie wird über den Spannungsregler mit 3,3 V versorgt [27].

2.2 Digitaler Signalprozessor TMS320C6713

Auf dem DSK-Bord befindet sich der digitale Signalprozessor „TMS320C6713“ der Firma *Texas Instruments* (*TI*). Hierbei handelt es sich um einen Floating-Point[1] Prozessor aus der „TMS320C67x“ Familie von *TI*. Da der DSP die Möglichkeit bietet, sowohl Fix- als auch Floating-Point Datentypen zu verarbeiten, lässt er sich sehr flexibel und vielfältig einsetzen.

Der Prozessor basiert auf der von *Texas Instruments* entwickelten „Very-Long-Instruction-Word“ (VLIW) Architektur. Dies ermöglicht dem Compiler, Befehle, die für den Prozessor parallel ausführbar sind, zu gruppieren und so für eine schnellere Abarbeitung dieser zu sorgen [26].
Dafür werden acht 32 Bit breite Befehle, gebündelt und als ein 256 Bit breites Paket, aus dem Programmspeicher geladen und anschließend den acht Recheneinheiten, nach einer Dekodierung des VLIWs, je einen Befehl zur weiteren Ausführung übergeben. Das LSB jedes 32 Bit-Befehls gibt an, ob die Instruktion im aktuellen Takt oder im nächsten Takt verwendet werden soll. Mit dieser Technik kann die CPU wahlweise einen Befehl oder bis zu acht Befehle parallel abarbeiten.

1 Fließkomma-Darstellung von Zahlenwerten

Der Prozessor ist bei einem Takt von 225 MHz in der Lage, zwei Multiplikationen pro Takt durchzuführen. Dies sind insgesamt bis zu 450 Millionen Multiplikationen (MACs) pro Sekunde [26].
Diese im Vergleich zu Mikrokontrollern hohe Leistungsfähigkeit ist auf die besondere Prozessorarchitektur zurückzuführen. Der Kern des Prozessors beinhaltet acht Rechenwerke. Diese sind unterteilt in zwei identische Datenpfade, „Data Path A“ und „Data Path B“, (s. Abbildung 2.3 [26]). Diese acht Rechenwerke beinhalten:

- vier arithmetische und logische Einheiten (ALUs), in welchen Fix- und Floating-Point Operationen durchgeführt werden können (.L1, .L2, .S1, .S2),
- zwei ALU´s für Fixpoint Operationen (.D1, .D2),
- zwei Multiplizierer, die sowohl Fix- als auch Floating-Point Berechnungen durchführen können (.M1, .M2).

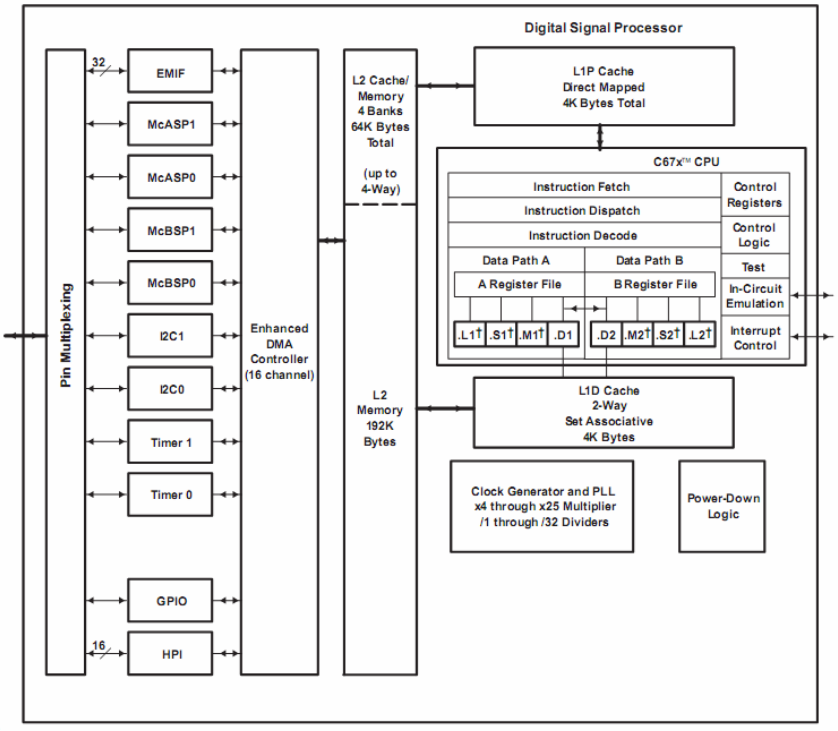


Abbildung 2.3 Interner Aufbau des DSP

Jeder dieser Pfade besitzt jeweils eine Einheit für Multiplikationen (.M), logische, arithmetische und Sprungoperationen (.L), (.S), sowie jeweils eine Einheit für den Datentransfer (.D). Fixpoint Additionen und Subtraktionen können ausgenommen von der Multiplikationseinheit (.M) von jeder Einheit ausgeführt werden. Jede Funktionseinheit hat direkten Lese- und Schreibzugriff auf das Register im eigenen Datenpfad. Des Weiteren gibt es einen Datenbus, der jede Funktionseinheit mit dem jeweils anderen Datenpfad verbindet. Dies ermöglicht eine

so genannte „Cross-Path“-Operation mit Daten aus dem jeweils andern Datenpfad. In jedem Datenpfad befinden sich sechzehn 32 Bit Register (A0 bis A15 und B0 bis B15), die unter den vier Funktionseinheiten im gleichen Datenpfad frei aufgeteilt werden können. Dabei ist zu beachten, dass nicht alle Register zu jeder Zeit zur Verfügung stehen. So befindet sich z.B. der „Stackpointer“ in Register B15 und die Rücksprungadresse in Register B3. Wenn diese Register benutzt werden, muss zuvor sichergestellt werden, dass die Daten nicht überschrieben werden.

Speicherorganisation des TMS320C6713 DSK

Der C6713 basiert auf einer Zwei-Ebenen-Cachespeicher (two-level cache) -Struktur. Diese besteht zum einen aus dem Level 1 Programmspeicher L1P (direkt verbunden) und dem Datenspeicher L1D mit jeweils 4 KByte Speichergröße, und zum anderen aus einem 256 KByte großen Level 2 Cachespeicher, der sich wie folgt zusammensetzt:

- 64 KByte L2 Speicher, der als Zwischenspeicher (Cache) oder adressierter Speicher verwendet werden kann
- 192 KByte L2 Speicher gehören zum SRAM.

Der Level 1 Programmspeicher besitzt einen 256 Bit breiten Bus zur CPU. Dies ermöglicht der CPU, in jedem Taktzyklus acht 32 Bit Befehle zu übergeben.
Der C6713 DSP besitzt ein 32 Bit „External Memory Interface“ (EMIF), welches eine nahtlose Verbindung zum externen Speicher (SDRAM, Flash, SBSDRAM, SRAM, EPROM) herstellt. Der DSP hat 512 MByte extern adressierbaren Speicher. Dieser kann „byteweise“ (8 Bit), „half-word“ (16 Bit) oder „word“ (32 Bit) -weise adressiert werden. Tabelle 2.1 zeigt die Standard-Speicherzuteilung auf dem „TMS320C6713 DSK“.

Tabelle 2.1 Speicherzuweisung des TMS320C6713

<table>
<tr><th>Adresse</th><th>C6713
Speicher Typ</th><th>C6713 DSK</th></tr>
<tr><td>0x00000000</td><td>Interner Speicher</td><td>Interner Speicher</td></tr>
<tr><td>0x00030000</td><td>Reserviert oder Peripherie</td><td>Reserviert oder Peripherie</td></tr>
<tr><td>0x80000000</td><td>EMIF CE0</td><td>SDRAM</td></tr>
<tr><td>0x90000000</td><td rowspan="2">EMIF CE1</td><td>Flash</td></tr>
<tr><td>0x90080000</td><td>CPLD</td></tr>
<tr><td>0xA0000000</td><td>EMIF CE2</td><td rowspan="2">Daughter Card</td></tr>
<tr><td>0xB0000000</td><td>EMIF CE3</td></tr>
</table>

Die CPU besitzt eine Vielzahl von Peripheriegeräten. Dazu gehören zwei Multichannel Audio Serial Ports (McASPs), zwei Multichannel Buffered Serial Ports (McBSPs), zwei Inter-Integrated Circuit (I^2C) Busse, ein General-Purpose Input/Output (GPIO) Modul, zwei frei nutzbare Timer, ein Host-Port Interface (HPI) und ein External Memory Interface (EMIF) als Schnittstelle zu den externen Speicherbausteinen und der angeschlossenen externen Peripherie [26].
Bedient werden die Peripheriegeräte vom „Enhanced Direct Memory Excess“ (EDMA) Controller (s. Abbildung 2.3).

Im Weiteren werden nur die für diese Arbeit benötigten Funktionen (McBSP und EDMA) näher erläutert. Für genauere Informationen über die weiteren Komponenten siehe [28]

2.2.1 AIC23 Codec

Der „TLV320AIC23“ (AIC23) ist ein Stereo-Codec zur Ein- und Ausgabe von Signalen über die angeschlossenen „Analog- Digital- Umsetzer“(ADU) und „Digital- Analog- Umsetzer“ (DAU), wobei das durch den ADU aufgenommene digitale Signal eine Repräsentation des analogen Eingangssignals darstellt.
Nachdem das digitale Signal im DSP verarbeitet worden ist, kann es auf dem umgekehrten Weg über den „Digital- Analog- Umsetzer“ wieder in ein analoges Ausgangssignal gewandelt werden. Ein Ausgangsfilter glättet dabei das Ausgangssignal. ADU, DAU und alle benötigten Filterfunktionen sind in dem integrierten „Single-Chip“ Codec AIC23 auf dem DSK Bord untergebracht.
AIC23 ist ein Stereo-Audio-Codec, basierend auf der „Sigma-Delta Technologie“[2].
Es ist möglich, Datenwörter der Länge 16, 20, 24 und 32 Bit zu übertragen. Ein Blockdiagramm des „TLV320AIC23“ und dessen Anbindung an das C6713DSK Bord ist im Anhang A [25] dargestellt.
Durch die im Vergleich zu anderen Anwendungen relativ niedrigen Frequenzen im Audiobereich kann durch Überabtastung des Signals ein großer Signal-Geräusch-Abstand (SNR) erreicht werden.
Der Codec arbeitet mit einem 12 MHz Takt, wodurch sich mit Überabtastraten von $250 \cdot F_S$ und $272 \cdot F_S$ exakt die Audio-Abtastrate 48 kHz (12 MHz/250) ergibt, sowie die für CDs übliche Abtastrate von 44,1 kHz (12 MHz/272).
Des Weiteren können die in Tabelle 2.2 [28] dargestellten Abtastfrequenzen im Abtastratenregister eingestellt werden [5].

2 eine Form der AD-Umsetzung oder DA-Umsetzung, welche sich von der Deltamodulation ableitet. Für nähere Informationen siehe Norsworthy,S: Delta-sigma data converters IEEE Press,1997.

Tabelle 2.2 AIC23 Abtastraten

Frequenz ID	Register Wert	Frequenz
DSK6713_AIC23FREQ_8KHZ	0x06	8000 Hz
DSK6713_AIC23FREQ_16KHZ	0x2c	16000 Hz
DSK6713_AIC23FREQ_24KHZ	0x20	24000 Hz
DSK6713_AIC23FREQ_32KHZ	0x0c	32000 Hz
DSK6713_AIC23FREQ_44KHZ	0x11	44100 Hz
DSK6713_AIC23FREQ_48KHZ	0x00	48000 Hz
DSK6713_AIC23FREQ_96KHZ	0x0e	96000 Hz

Der ADU wandelt ein analoges Eingangssignal in ein äquivalentes, diskretes, digitales Datenwort um. Dieser Wert repräsentiert das Eingangssignal zu einem Abtastzeitpunkt. Für die Ausgabe über den DAU werden die gewandelten Werte über einen Interpolationsfilter ausgegeben.
ADU und DAU erreichen durch die Überabtastung und Rauschformung (noise shaping) des „Sigma-Delta Umsetzers“ einen SNR von bis zu 90 dB [25].
Die Kommunikation zwischen dem Codec und der CPU für die Ein- und Ausgabe der Signale erfolgt über zwei mehrkanalige, gepufferte, serielle Ports (McBSPs).
Das Blockdiagramm mit allen Ein- und Ausgängen des AIC23-Codec ist in Abbildung 2.4 dargestellt [27].

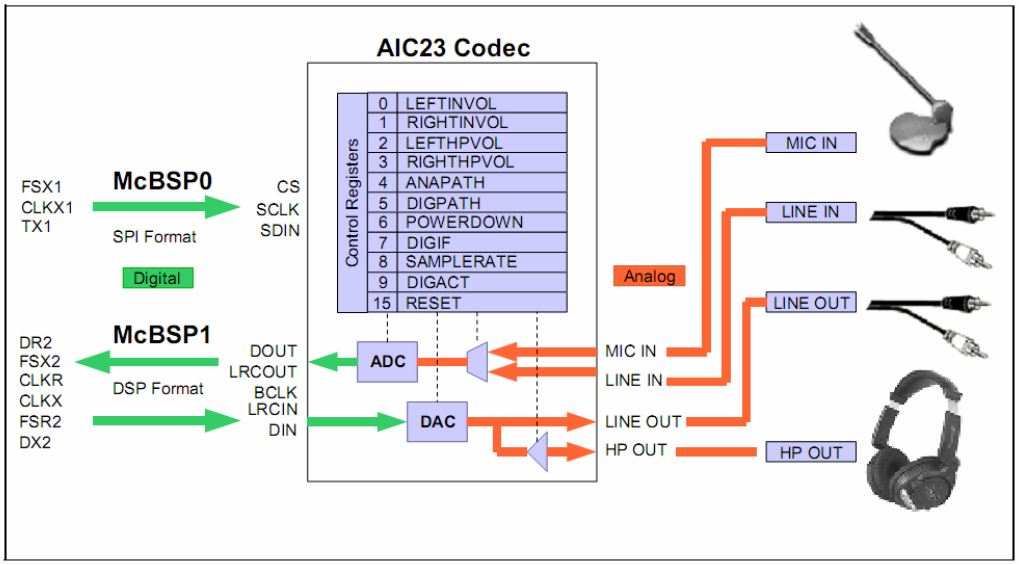


Abbildung 2.4 Block Diagramm AIC23 Codec

Der AIC23 verfügt über zwei Eingangskanäle (linker und rechter Audio-Kanal). Beide Kanäle besitzen eine separat einstellbare Eingangsverstärkung. Diese kann logarithmisch von 12 dB bis -34.5 dB eingestellt werden. Die Aussteuergrenze des ADU beträgt 1 V_{RMS} bei einer nominellen Betriebsspannung von 3,3 V. Um Verzerrungen zu vermeiden, ist es wichtig, die-

se Spannung nicht zu überschreiten. Durch die in Abbildung 2.5 dargestellte Schaltung wird die Spannung am „LINE IN“ Eingang halbiert. Damit darf die maximal anliegende Spannung $2\ V_{RMS}$ nicht überschreiten.

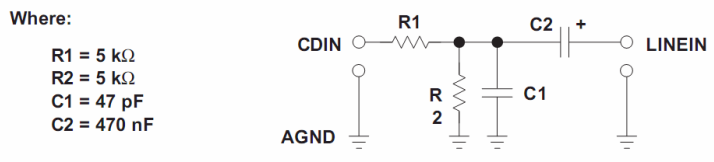


Abbildung 2.5 Schaltung am „LINE IN“ Eingang

Die maximale Ausgangsspannung des DAUs beträgt ebenfalls 1 V_{RMS}. Die beiden Ausgänge sind in der Lage eine Last von $10\,k\Omega$ und $50\,pF$ zu treiben.

2.2.2 Multichannel Bufferd Serial Ports

Auf dem Bord sind zwei „Multichannel Bufferd Serial Ports“ (McBSPs) vorhanden (s. Abbildung 2.6 [26]). Diese bilden die Schnittstelle zu den angeschlossenen Peripheriegeräten. Die McBSPs können im „full duplex“ Modus mit unabhängigem Takt und einstellbarer Rahmengröße (für das Senden und Empfangen) mit den angeschlossenen Komponenten kommunizieren. Dabei sind unterschiedliche Datengrößen zwischen 8 und 32 Bit einstellbar. Des Weiteren ermöglichen die McBSPs eine Kommunikation mit externen Geräten, während intern Daten verarbeitet werden können (s. Abbildung 2.7 [23]).

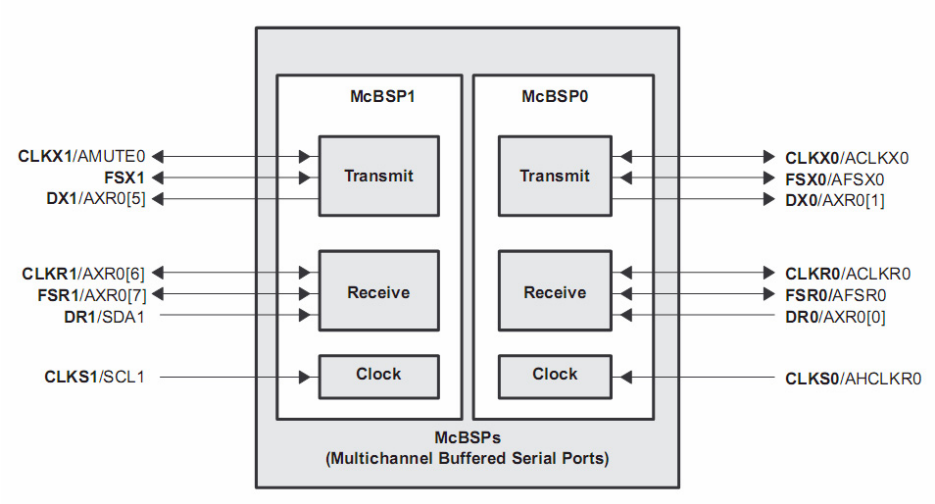


Abbildung 2.6 Block Diagramm des McBSP

Das Senden der Daten erfolgt über den DX (Data Transmit) Pin, während das Empfangen über DR (Data Recive) abgewickelt wird. Die CPU oder der DMA Controller können Daten direkt aus dem Datenempfangsregister DRR (Data Recive Register) lesen oder zum Senden der Daten in das Datensenderegister DXR (Data Transmit Register) schreiben. Die Verbin-

dung zwischen Datensenderegister und DX erfolgt über das Sendeschreiberegister XSR (Transmit Shirft Register), welches die zu sendenden Daten an den DX-Pin übergibt.

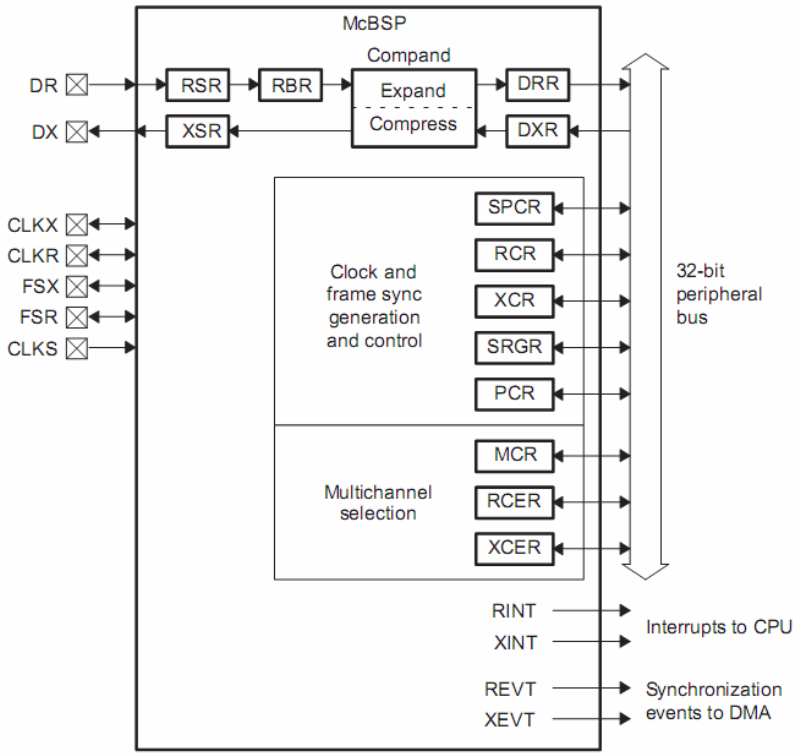


Abbildung 2.7 Aufbau eins McBSP Kanals

Das Empfangsschieberegister RSR (Recive Shift Register) kopiert die über DR empfangenen Daten in das Empfangspufferregister RBR (Recive Buffer Register). Anschließend werden die Daten an das Empfangsregister DRR weitergegeben. Dort können sie von der CPU oder dem DMA Controller gelesen werden [5].

2.2.3 Enhanced Direct Memory Access

Der „TMS320C6713“ besitzt einen integrierten Controller für den direkten Speicherzugriff „Enhanced Direct Memory Access Controller“ (EDMA) [28]. Dieser ist in der Lage, den gesamten 32 Bit Adressraum des DSPs unabhängig von der CPU zu kontrollieren. Dies ermöglicht einen Datentransfer zwischen den angeschlossenen Komponenten zum internen Speicher über den EDMA Controller [22].
Dies hat den Vorteil, dass die CPU nicht auf externe Ereignisse, wie z.B. einen „Interrupt“ eines McBSP Kanals, reagieren muss und demzufolge einen anderen Prozess unterbricht.

Der EDMA Controller des „TMSC6713“ hat 16 Kanäle, mit denen bis zu 16 Übertragungen, ohne das Einbeziehen der CPU, durchgeführt werden können. Zu jedem Kanal gibt es einen

Parameter RAM (PaRAM), in dem die zahlreichen Funktionen des EDMA parametriert werden. Abbildung 2.8 zeigt den Aufbau des PaRAM [19].
Bevor eine Datenübertragung mit Hilfe des EDMA Controllers gestartet werden kann, muss diese von der CPU initialisiert und gestartet werden. Dies geschieht über den PaRAM.
Kanäle, die nicht direkt für eine Datenübertragung verwendet werden, können als „Link"-Eingang einer Übertragung genutzt werden. Dies ermöglicht der CPU „Ping-Pong"- oder verkettete Buffer einzurichten. „Ping-Pong" Buffer bieten die Möglichkeit, Daten aus einem Buffer zu verarbeiten, während der jeweils andere Buffer dem EDMA zum Einlesen oder Ausgeben der Daten zur Verfügung steht.

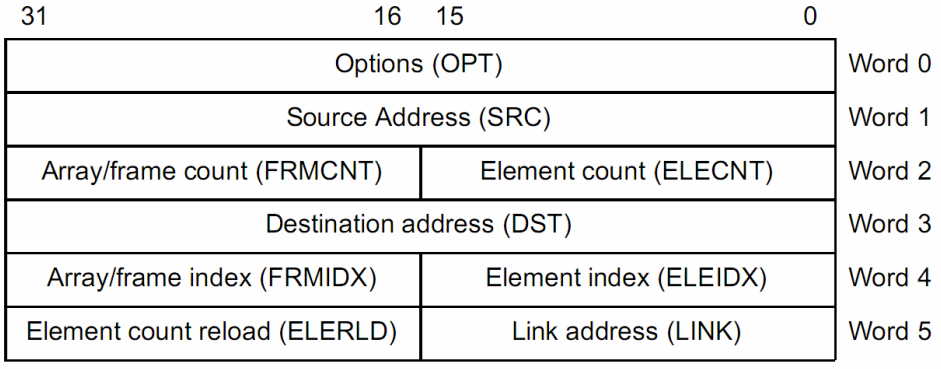


Abbildung 2.8 Aufbau des EDMA Parameter RAM

Register im PaRAM

Option Register (OPT): In diesem Register werden alle Optionen, die der EDMA zur Verfügung stellt, konfiguriert. Unter anderem kann eingestellt werden, ob es sich bei den zu übertragenden Daten um 1D oder 2D Arrays handelt. Außerdem kann die Priorität, mit welcher der jeweilige Kanal arbeitet, die Inkrementierungsart und die Größe des zu übertragenen Datentyps (8, 16, 32 Bit) festgelegt werden.

Source Adress Register (SRC): In diesem Register wird die Adresse eingetragen, von welcher die Daten gelesen werden.

Array/Fram Count Register (FRMCNT): In dieses Register wird ein 16 Bit Wert eingetragen, welcher die Anzahl der Frames (für 1D Transfer) oder die Anzahl der Arrays (für 2D Transfer) in einem Block festlegt.

Element Count Register (ELECNT): Legt die Anzahl der Elemente pro Frame fest (für 1D Transfer) oder die Anzahl der Elemente in einem Array (für 2D Transfer). Der Wert für Element Count kann zwischen 1 und 65535 liegen.

Destination Address Register (DST): In diesem Register wird die Zieladresse der EDMA Datenübertragung eingetragen (Speicherbereich, in dem die gelesenen Daten gespeichert werden).

Index Parameter (IDX): Der Index in der EDMA- Parameter-Liste spezifiziert den Array/Frame und Element-Index, welche zur Adressmodifikation benötigt werden. Der EDMA benutzt die Indices für ein Adressupdate abhängig vom eingestellten Modus (1D oder 2D).

Array/Fram Index Register (FRMIDX): Dies ist ein 16 Bit Wert, der den „Adressoffset“ bis zum nächsten Array/Frame angibt. Dieser kann zwischen –32768 und 32767 liegen.

Element Index (ELEIDX): Dies ist ebenfalls ein 16 Bit Wert, der den „Adressoffset“ zum nächsten Element angibt.

Element Count Reload (ELERLD): Lädt einen neuen Wert in das Element Count Register (ELECENT), sobald das letzte Element eines „Frames” übertragen wurde. Diese Funktion wird nur im 1D Betrieb benötigt.

Link Adress (LINK): In diesem Register befindet sich die Adresse der Parameterliste, welche als Nächstes geladen wird. Dieses Register wird benötigt, wenn im „Option Register“ die Funktion Link aktiviert ist oder wenn die Datenübertragung beendet werden soll. Um die Übertragung zu beenden, wird ein „NULL-Pointer“ in das Register geschrieben.

Ablauf einer Datenübertragung

Die Funktionsweise des EDMA wird im Folgenden an dem Beispiel eines kontinuierlichen Lesens eines McBSP Kanals erklärt. Dafür ist es notwendig, die Parameterliste so zu konfigurieren, dass diese mit sich selbst „verlinkt“ ist.
„Enhanced Direct Memory Access“ ist in zwei Einheiten aufgeteilt (s. Abbildung 2.9 [22]), den Transfer Controller (EDMATC) und den Chanel Controller. Der Chanel Controller startet den Datentransfer, ausgelöst von einem Signal der CPU oder einem Interrupt. Daraufhin lädt der Controller die zuvor für den Datentransfer über McBSP konfigurierte Parameterliste. Der Transfer Controller sendet die Parameterliste an den McBSP, damit dieser die Daten, wie in der Parameterliste beschrieben, sendet. Daraufhin beginnt das Senden der Daten an den EDMA Controller. Ist ein Transfer abgeschlossen, sendet der Transfer Controller den „Transfer Complete Code“ (TCC) an den Chanel Controller. Dieser löst daraufhin den EDMA Interrupt aus. Dadurch wird der CPU mitgeteilt, dass ein neues Datenpaket zur Verfügung steht. Die Adresse des Datenpaketes wurde vorher durch die Parameterliste festgelegt und ist daher der CPU bekannt. Anschließend lädt der Chanel Controller die Parameterliste, die unter der

Link-Adresse angegeben ist. Da die Parameterliste in diesem Beispiel für eine kontinuierliche Übertragung mit sich selbst verlinkt ist, lädt er die gleiche Liste. So kommt eine kontinuierliche Datenübertragung zustande.
Des Weiteren ist es möglich, durch die Link-Adresse einen anderen EDMA-Kanal zu aktivieren. Zeigt die Link-Adresse auf eine leere Parameterliste, wird die Übertragung beendet.

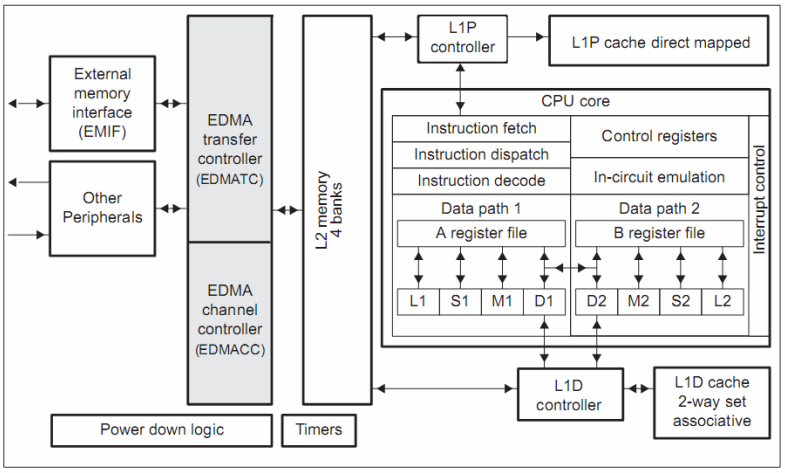


Abbildung 2.9 CPU Blockdiagramm mit EDMA

2.3 Code Composer Studio

Code Composer Studio (CCS) bietet eine integrierte Entwicklungsumgebung (IDE) für Echtzeit DSP-Anwendungen, basierend auf der Programmiersprache C. Diese Entwicklungsumgebung beinhaltet einen C-Compiler, Assembler und Linker. Des Weiteren bietet CCS zahlreiche Möglichkeiten der grafischen Darstellung sowie des Echtzeit- „Debugging“.
CCS bietet zwei Möglichkeiten der Codebeschreibung. Entweder in C oder in der hardwarenahen Assembler-Programmiersprache. Es ist außerdem möglich, eine Kombination aus beiden einzusetzen (Linear Assembler).
Der in CCS integrierte C-Compiler wandelt den C-Code zunächst in einen Assembler-Code um. Dieser wird dann vom „Assembler“ in den ausführbaren Maschinencode mit der Endung .obj umgewandelt. Anschließend wird diese Datei vom „Linker“ mit der Objektbibliothek zusammengeführt und als ausführbares Programm mit der Dateiendung .out gespeichert. Diese Datei kann anschließend auf den Prozessor geladen und dort ausgeführt werden (s. Abbildung 2.10 [20]).
Zu den vielen „Debugging“-Möglichkeiten von CCS gehören neben Echtzeitdebugging das Beobachten und Manipulieren von Variablen und Registerinhalten sowie die grafische Anzeige von Variablen zur Laufzeit. Darüber hinaus besteht die Möglichkeit, Haltepunkte zu setzen, sowie die Laufzeitmessung (Profiling) einzelner Programmabschnitte oder Funktionen durchzuführen [4].

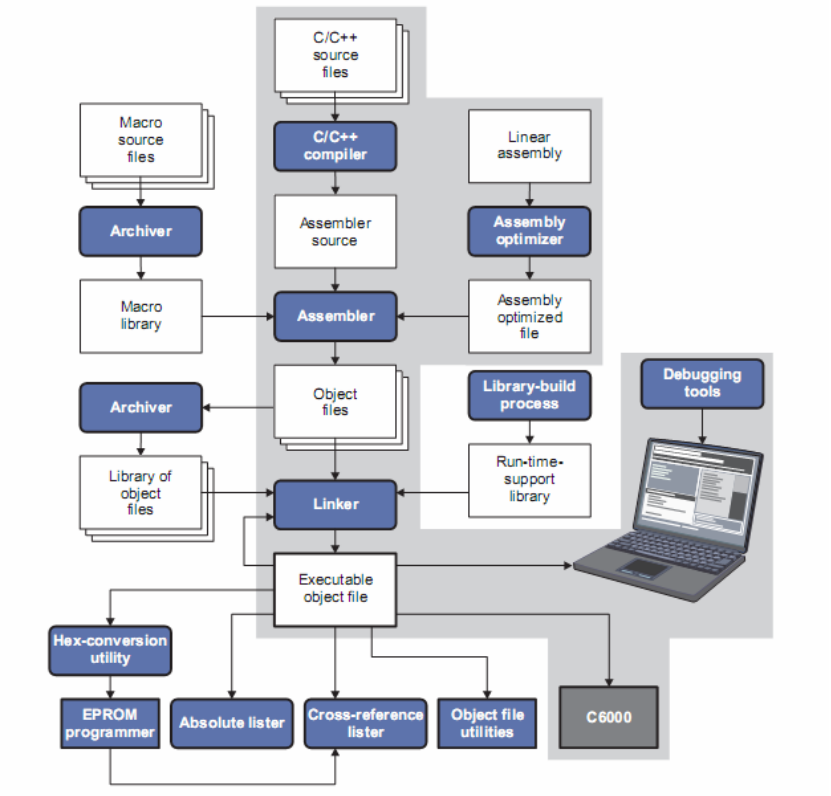


Abbildung 2.10 CCS Flussdiagramm

3 Untersuchung der FFT Algorithmen

Die schnelle Fourier-Transformation (englisch fast Fourier transform, FFT abgekürzt) ist ein Algorithmus zur effizienten Berechnung der Werte einer diskreten Fourier-Transformation (DFT). Bei diesem Algorithmus handelt es sich um Teile- und Herrsche- Verfahren[3]. Im Gegensatz zur direkten Berechnung verwendet eine schnelle Fourier-Transformation zuvor berechnete Zwischenergebnisse und spart dadurch arithmetische Rechenoperationen ein. Die Entwicklung des Verfahrens wird James Cooley und John W. Tukey zugeschrieben, die dieses 1965 veröffentlichten (Später wurde entdeckt, dass eine Form des Algorithmus bereits 1805 von Carl Friedrich Gauß zur Berechnung von Asteroidenflugbahnen eingesetzt wurde).
Darüber hinaus wurden eingeschränkte Formen des Algorithmus mehrfach vor Cooley und Tukey entwickelt, so z. B. von Irving John Good (1960). Später gab es zahlreiche Verbesserungsvorschläge und Variationen des Algorithmus. So etwa von Georg Bruun, C. M. Rader und Leo I. Bluestein [3].

3.1 Anwendungsbereiche

Die Fourier-Transformation findet in vielen Themenkreisen ihre Anwendung. E. Brigham spricht in seinem Buch „Schnelle Fourier-Transformation“ von der Allgegenwärtigkeit der Fourier-Transformation (FT).
„Das Wort ‚allgegenwärtig’ bedeutet, zugleich überall zu sein. Wegen der großen Vielzahl unterschiedlicher Themenkreise, die mit Hilfe der Fourier-Transformation bearbeitet werden können, ist die Benutzung des Attributs ‚allgegenwärtig’ für die Fourier-Transformation sicherlich angebracht“ (Brigham 1995, Seite 20).
Beispiele hierfür sind:

Lineare Systeme: Die Fourier-Transformierte des Ausgangssignals eines linearen Systems ist gleich dem Produkt der Übertragungsfunktion und der Fourier-Transformierten des Eingangssignals [8].

Antennen: Das Strahlungsdiagramm einer Antenne ergibt sich als Fourier-Transformierte ihrer Flächenstromverteilung [11].

Optik: Die Amplitudenverteilung des Lichtes auf der vorderen und der hinteren Brennebene einer konvexen Linse folgt der Beziehung der Fourier-Transformierten [1].

3 „Teile und Herrsche“ findet in vielen Teilgebieten der Informatik Anwendung und beschreibt einen reduktonistischen Lösungsansatz [6]

Stochastische Signale: Das Leistungsdichtespektrum eines stochastischen Signals ist die Fourier-Transformierte der Autokorrelationsfunktion [12].

Randwertproblem: Partielle Differentialgleichungen lassen sich mit Hilfe der Fourier-Transformation lösen [2].

Signalverarbeitung: In der Signalverarbeitung entspricht die Fourier-Transformierte eines Signals dessen Spektrum im Frequenzbereich.

Faltung: Eine Faltung im Zeitbereich entspricht einer Multiplikation im Frequenzbereich. Mit Hilfe der Fourier-Transformation lassen sich die Zeitbereichssignale in den Frequenzbereich transformieren, dort multiplizieren und anschließend durch die inverse Fourier-Transformation zurück transformieren.

Obwohl die genannten Beispiele aus unterschiedlichsten Anwendungsbereichen stammen, folgen sie doch alle der Theorie der Fourier-Transformation.

3.2 Sande- Tukey Algorithmus

Das Prinzip der schnellen Fourier-Transformation (FFT) beruht auf dem Gedanken, die zu berechnenden Summen der Länge *N* auf Summen gleicher Struktur mit der Länge *N/2* zu reduzieren [13]. Wie aus der Diskreten Fourier-Transformation bekannt, gilt:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-jkn\frac{2\pi}{N}} \quad \text{für } k = 0,1,2,...N-1 \tag{3.1}$$

Für den Sande-Tukey Algorithmus lässt sich die obige Summe am Einheitskreis als Summe gegenüberliegender Therme wie folgt schreiben:

$$X_{2k} = \sum_{n=0}^{N-1} x_n \cdot e^{-j2kn\frac{2\pi}{N}} = \sum_{n=0}^{(N/2)-1} \left(x_n \cdot e^{-j2kn\frac{2\pi}{N}} + x_{n+N/2} \cdot e^{-j2k(n+N/2)\frac{2\pi}{N}} \right) \tag{3.2}$$

Daraus folgt durch Rechenregeln der Exponentialfunktion:

$$X_{2k} = \sum_{n=0}^{(N/2)-1} \left(x_n + x_{n+N/2} \right) \cdot e^{-jkn\frac{2\pi}{N/2}} \tag{3.3}$$

Mit $x'_n = x_n + x_{n+N/2}$ ergibt sich:

$$X_{2k} = \sum_{n=0}^{(N/2)-1} x'_n \cdot e^{-jkn\frac{2\pi}{N/2}} \tag{3.4}$$

Somit lassen sich die Koeffizienten mit geradem Index als Summe der halben Länge darstellen, wobei die daraus entstandene Summe von der Art gleich der ursprünglichen Summe ist.
Die gleiche Überlegung lässt sich für die ungeraden Koeffizienten anstellen. Somit folgt aus:

$$X_{2k+1} = \sum_{n=0}^{N-1} x_n \cdot e^{-j(2k+1)n\frac{2\pi}{N}} \tag{3.5}$$

$$X_{2k+1} = \sum_{n=0}^{(N/2)-1} \left((x_n - x_{n+N/2}) \cdot e^{-jn\frac{2\pi}{N}} \right) \cdot e^{-jkn\frac{2\pi}{N/2}} \tag{3.6}$$

Mit $x''_n = (x_n - x_{n+N/2}) \cdot e^{-jn\frac{2\pi}{N}}$ ergibt sich:

$$X_{2k+1} = \sum_{n=0}^{(N/2)-1} x''_n \cdot e^{-jkn\frac{2\pi}{N/2}} \tag{3.7}$$

D.h. auch die Koeffizienten mit ungeradem Index lassen sich als Summe der halben Länge darstellen.
Der oben beschriebene Reduktionsvorgang lässt sich beliebig weiterführen, solange, bis die Summen nur noch aus einem einzigen Summanden bestehen [13]. Da bei diesem Verfahren die Anzahl der zu berechnenden Daten im Frequenzbereich bei jedem Schritt halbiert wird, ist dieser Algorithmus auch unter dem Namen „Dezimation im Frequenzbereich“ (engl. dezimation in frequency) oder kurz DIF bekannt.
Bei diesem Verfahren zur Lösung der FFT findet eine zyklische Vertauschung der Eingangsbits statt (Beweis siehe [13] Seite 194ff). Dies hat zur Folge, dass der Ergebnisvektor X_k in umgekehrter Bitreihenfolge (Bit Reversal) vorliegt. Dieser Zusammenhang wird in Abbildung 3.1 a dargestellt.

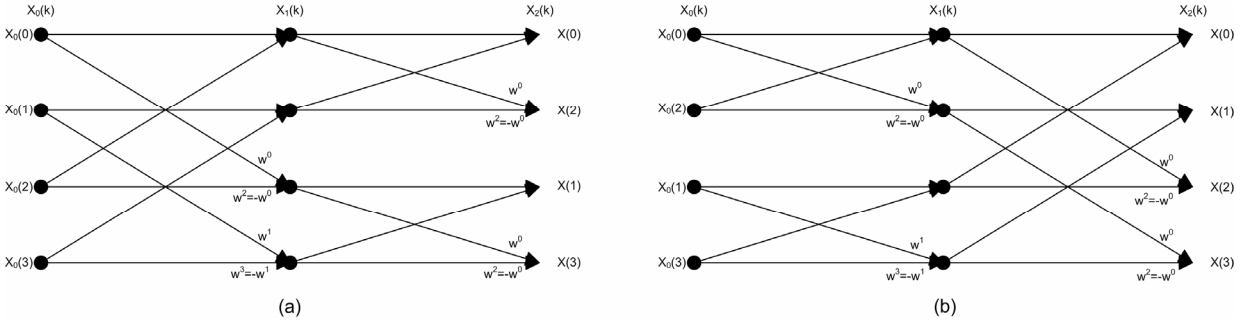


Abbildung 3.1 Sande-Tukey Signalflussgraph: a) Eingangswerte in natürlicher Reihenfolge, b) Eingangswerte in bit- gespiegelter Reihenfolge

Um die Fourier-Koeffizienten in natürlicher Reihenfolge zu erhalten, muss die Dualzahldarstellung der Vektor-Indices gespiegelt werden. Zudem besteht die Möglichkeit, den Eingangsvektor in bitgespiegelter Reihenfolge zur Verfügung zu stellen (vgl. Abbildung 3.1 b)

3.3 Cooley und Tukey Algorithmus

Wie zuvor beim Sande-Tukey Algorithmus beschrieben, beruht auch der Cooley und Tukey Algorithmus auf dem Gedanken, die zu berechnenden Summen der Länge *N* auf Summen gleicher Struktur mit der Länge *N/2* zu reduzieren.
Bei diesem Verfahren zur Berechnung der FFT Koeffizienten werden die Eingangswerte in zwei Summen mit der Länge *N/2* zerlegt.

$$X_k = \sum_{n=0}^{(N/2)-1} x_{2n} \cdot e^{-j2nk\frac{2\pi}{N}} + \sum_{n=0}^{(N/2)-1} x_{2n+1} \cdot e^{-j(2n+1)k\frac{2\pi}{N}} \tag{3.8}$$

Daraus folgt:

$$X_k = \sum_{n=0}^{(N/2)-1} x_{2n} \cdot e^{-jnk\frac{2\pi}{N/2}} + e^{-jk\frac{2\pi}{N}} \cdot \sum_{n=0}^{(N/2)-1} x_{2n+1} \cdot e^{-jnk\frac{2\pi}{N/2}} \tag{3.9}$$

Substituiert man $k = N/2 + l$ mit $l = 0,1,...,N/2-1$, ergibt sich aus Gleichung (3.9)

$$X_{N/2+l} = \sum_{n=0}^{(N/2)-1} x_{2n} \cdot e^{-jn(N/2+l)\frac{2\pi}{N/2}} + e^{-j(N/2+l)\frac{2\pi}{N}} \cdot \sum_{n=0}^{(N/2)-1} x_{2n+1} \cdot e^{-j(N/2+l)\frac{2\pi}{N/2}} \tag{3.10}$$

Mit $e^{-j\pi} = -1$ ergibt sich:

$$X_{N/2+l} = \sum_{n=0}^{(N/2)-1} x_{2n} \cdot e^{-jnl\frac{2\pi}{N/2}} - e^{-jl\frac{2\pi}{N}} \cdot \sum_{n=0}^{(N/2)-1} x_{2n+1} \cdot e^{-jnl\frac{2\pi}{N/2}} \tag{3.11}$$

Dies entspricht einer halben Drehung auf dem Einheitskreis. Wird nun $l = k$ gesetzt und werden die geraden Koeffizienten in eine Summe und die ungeraden Koeffizienten in eine zweite Summe zusammengefasst, ergibt sich für die geraden Koeffizienten

$$X'_k = \sum_{n=0}^{(N/2)-1} x_{2n} \cdot e^{-jnk\frac{2\pi}{N/2}} \tag{3.12}$$

und für die ungeraden Koeffizienten

$$X''_k = \sum_{n=0}^{(N/2)-1} x_{2n+1} \cdot e^{-jnk\frac{2\pi}{N/2}} \tag{3.13}$$

Die Gleichungen (3.9) und (3.11) lassen sich für $k = 0,1,...,N/2-1$ und mit (3.12), (3.13) wie folgt schreiben:

$$\begin{aligned} X_k &= X'_k + e^{-jk\frac{2\pi}{N}} \cdot X''_k \\ X_{N/2+k} &= X'_k - e^{-jk\frac{2\pi}{N}} \cdot X''_k \end{aligned} \tag{3.14}$$

Wie in Gleichung (3.14) zu sehen, unterscheiden sich die beiden Therme nur im Vorzeichen. Durch diese Art der Zerlegung ist es ausreichend, einen der beiden Therme zu berechnen, da sich der jeweils andere Ausdruck daraus ergibt.
Diese Aufteilung in gerade und ungerade Koeffizienten lässt sich analog zum Sande-Tukey Algorithmus fortführen, bis die Summe nur noch aus einem einzelnen Summanden besteht.

Abbildung 3.2 zeigt ein Beispiel für diesen Algorithmus. In Abbildung 3.2 a werden die Eingangswerte in natürlicher Reihenfolge bereitgestellt. Dies hat zur Folge, dass der Ergebnisvektor, wie auch beim Sande-Tukey Algorithmus, in bit-gespiegelter Reihenfolge vorliegt. Alternativ hierzu können auch die Eingangswerte in bit-gespiegelter Reihenfolge bereitgestellt werden (s. Abbildung 3.2 b).

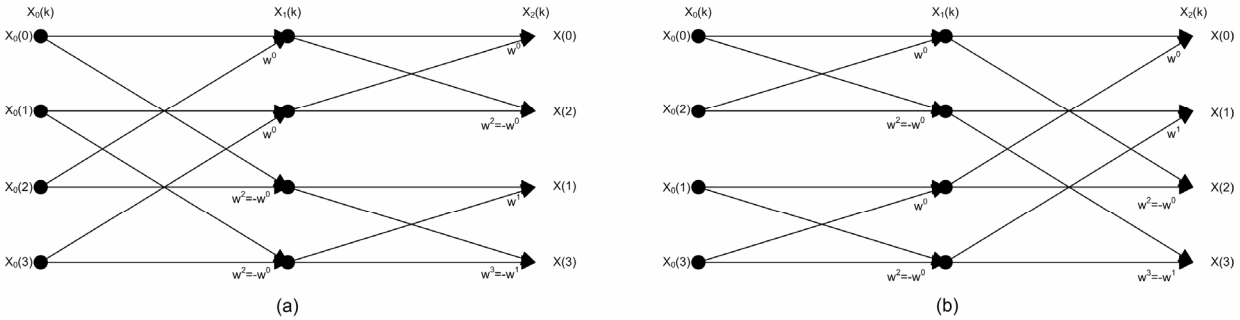


Abbildung 3.2 Cooley und Tukey Signalflussgraph: a) Eingangswerte in natürlicher Reihenfolge, b) Eingangswerte in bit- gespiegelter Reihenfolge

Dieser Algorithmus wird in der Literatur oft mit dem Namen „Dezimation im Zeitbereich“ (engl. decimation in time) oder kurz DIT bezeichnet. Diese Bezeichnung kommt von einer Herleitung des Algorithmus, der sich auf das Konzept der Abtastraten–Reduktion bzw. das Weglassen von Abtastwerten stützt. Daher der Begriff „Dezimation im Zeitbereich“ [3].

3.4 Radix 2 FFT

Der Radix2 oder auch Basis 2 Algorithmus ist ein mathematisches Lösungsverfahren der FFT. Hierbei wird, wie in Kapitel 3.2 und Kapitel 3.3 beschrieben, eine Zerlegung der DFT in zwei DFTs der halben Länge durchgeführt. Dies kann sowohl als Dezimation im Zeitbereich wie auch im Frequenzbereich ausgeführt werden. Dabei muss die Anzahl der Eingangswerte einer Potenz der Basis 2 entsprechen. Im Folgenden werden die Schritte zur Durchführung einer möglichst effizienten FFT anhand der Dezimation im Zeitbereich erklärt. Die Dezimation im Frequenzbereich erfolgt wie in Kapitel 3.2 beschrieben analog dazu.
Durch die Zerlegung einer *N*-Punkte Diskreten Fourier-Transformation in eine Summe von zwei $N/2$ DFTs ergibt sich für $e^{-j2\pi/N} = W_N$ (Twiddle-Faktor) und $k = 0,1,2,...,N-1$ aus Gleichung (3.9)

$$X_k = \sum_{n=0}^{(N/2)-1} x_{2n} \cdot W_{N/2}^{kn} + W_N^k \cdot \sum_{n=0}^{(N/2)-1} x_{2n+1} \cdot W_{N/2}^{kn} \tag{3.15}$$

wobei in x_{2n} die geraden Indexwerte und in x_{2n+1} die ungeraden Indexwerte der DFT enthalten sind. Jede der beiden DFTs muss lediglich für $N/2$ von $k = 1,2,...,N/2-1$ ausgewertet werden, da $W_{N/2}^{kn}$ periodisch mit der Periode $N/2$ ist (vgl. Herleitung in Kapitel 3.3 und Abbildung 3.3 [10]).
Da die Anzahl der Eingangswerte einer Potenz der Basis 2 entspricht, lassen sich die Summen aus Gleichung (3.15) wiederum in jeweils zwei Summen mit den geraden und ungeraden

Koeffizienten aufteilen. Dies kann solange fortgeführt werden, bis die jeweiligen Summen aus lediglich einem Summanden bestehen.
Bei reellen Eingangswerten für $x(n)$ ist der Realteil von $X(f)$ symmetrisch um $N/2$ und der Imaginärteil antisymmetrisch (punktsymmetrisch) um $N/2$. Anders ausgedrückt: Der Realteil von $X(f)$ ist eine gerade Funktion und der Imaginärteil ist eine ungerade Funktion. Daraus folgt, dass die Fourier-Koeffizienten zwischen $N/2$ und $N-1$ als „negative Frequenz“ interpretiert werden können.

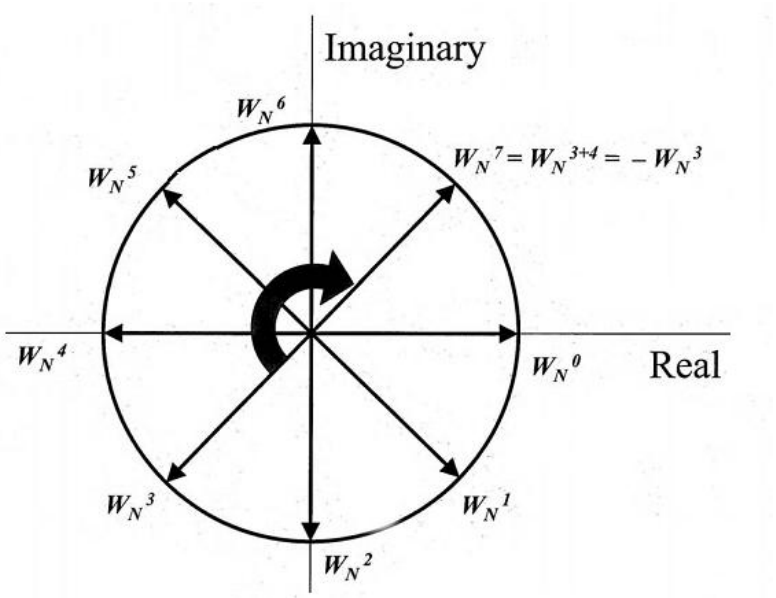


Abbildung 3.3 Darstellung der Twiddle-Faktoren im Einheitskreis

Zur Veranschaulichung der einzelnen Rechenschritte wird der Ablauf der Berechnung in einem Signalflussgraphen dargestellt. Die Darstellung erfolgt mit so genannten „Butterflys“, mit denen die durchzuführenden Rechenoperationen grafisch dargestellt werden (Abbildung 3.4 [10]).

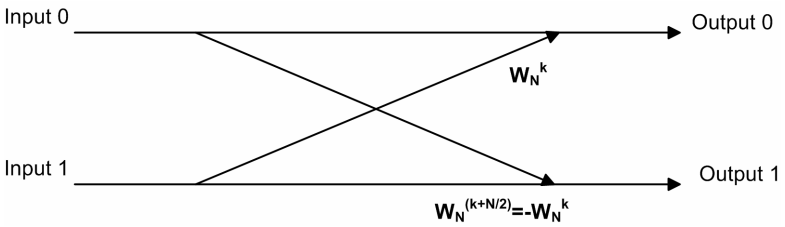


Abbildung 3.4 Radix 2 Butterfly

Mit Hilfe des Butterflys können die Ausgangswerte Output 1 und Output 2 wie folgt berechnet werden. Der erste Ausgangswert ergibt sich aus Input 1, addiert mit dem Ergebnis der Multiplikation aus Input 2 und dem Twiddle-Faktor W_N^k. Da $W_N^{(k+N/2)} = -W_N^k$ gilt, ergibt sich Output 2 aus Input 1 und der Subtraktion der zuvor berechneten Multiplikation von Output 2

und W_N^k. Somit sind für die Berechnung des Butterflys nur eine komplexe Multiplikation sowie eine Subtraktion und Addition notwendig. Durch die Periodizität von W_N^k mit *N/2* reduziert sich die Anzahl der benötigten Multiplikationen um den Faktor zwei.
Mit Hilfe der Butterflys lässt sich die systematische Berechnung der FFT grafisch darstellen. Abbildung 3.5 zeigt ein Beispiel für die Berechnung einer Radix 2 FFT mit Dezimation im Zeitbereich und Eingangswerten in bit-gespiegelter Reihenfolge (vgl. Kapitel 3.3).

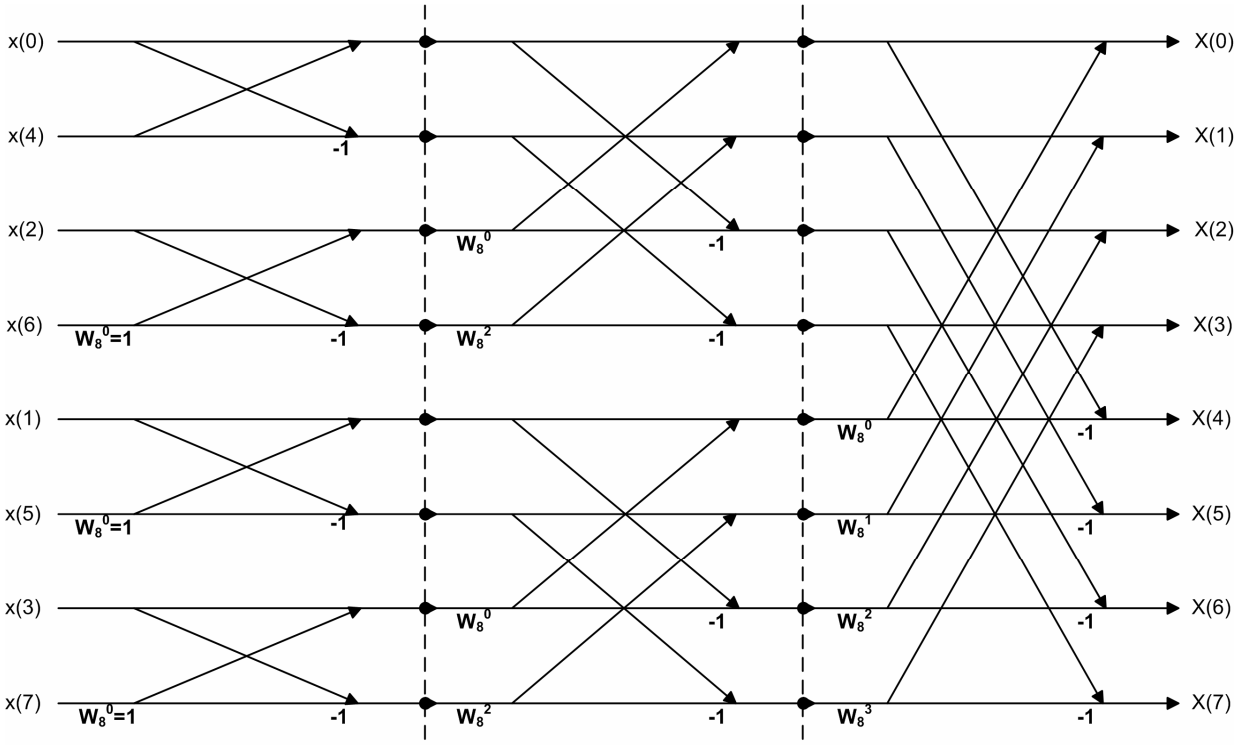


Abbildung 3.5 Radix 2 FFT DIT

Die Anzahl der zur Berechnung notwendigen Stufen ergibt sich aus der Anzahl der Eingangswerte. In diesem Beispiel sind für 2^3=8 Eingangswerte 3 Stufen notwendig. In den einzelnen Stufen erfolgt die Berechnung entsprechend Abbildung 3.4. Das Ergebnis der Berechnung sind die Koeffizienten der Fourier-Transformierten des Eingangssignals.
Für die Berechnung der Koeffizienten ergeben sich somit 12 komplexe Multiplikationen sowie 24 komplexe Additionen. Zum Vergleich: Eine DFT Berechnung mit dem gleichen Ergebnis erfordert 64 komplexe Multiplikationen und 56 Additionen ($N \cdot (N-1)$).

Allgemein ergibt sich für den Rechenaufwand einer Radix 2 FFT:

$\frac{N}{2} \cdot \log_2 N$ komplexe Multiplikationen,

$N \cdot \log_2 N$ komplexe Additionen.

Bei genauer Betrachtung der Twiddle-Faktoren lässt sich eine weitere Reduzierung der Multiplikationen durchführen. In Abbildung 3.5 ist in der ersten Stufe der Twiddel-Faktor $W_8^0 = 1$. Somit lässt sich diese Stufe ohne Multiplikation berechnen.

3.5 Radix 4 FFT

Die Überlegungen zur Radix 2 FFT lassen sich weiter fortführen, indem die ursprüngliche DFT in vier DFTs mit jeweils einem Viertel der Länge aufgeteilt wird. Daher ergeben sich für die Radix 4 FFT vier Summen mit jeweils einem Viertel der Werte.

$$X_k = \sum_{n=0}^{(N/4)-1} x_{4n} \cdot e^{-j4nk\frac{2\pi}{N}} + \sum_{n=0}^{(N/4)-1} x_{4n+1} \cdot e^{-j(4n+1)k\frac{2\pi}{N}} + \sum_{n=0}^{(N/4)-1} x_{4n+2} \cdot e^{-j(4n+2)k\frac{2\pi}{N}} + \sum_{n=0}^{(N/4)-1} x_{4n+3} \cdot e^{-j(4n+3)k\frac{2\pi}{N}} \tag{3.16}$$

Mit $e^{-j42\pi/N} = W_{N/4}$ und durch Umstellen der Gleichung ergibt sich daraus:

$$X_k = \sum_{n=0}^{(N/4)-1} x_{4n} W_{N/4}^{kn} + W_N^k \sum_{n=0}^{(N/4)-1} x_{4n+1} W_{N/4}^{kn} + W_N^{2k} \sum_{n=0}^{(N/4)-1} x_{4n+2} W_{N/4}^{kn} + W_N^{3k} \sum_{n=0}^{(N/4)-1} x_{4n+3} W_{N/4}^{kn} \tag{3.17}$$

Die Eingangswerte müssen dabei einer Potenz der Basis 4 entsprechen, um diese wie in Gleichung (3.17) zerlegen zu können.
Die Berechnung der Radix 4 Butterflys erfolgt analog zu der Berechnung der Radix 2 Butterflys. Dabei ist zu beachten, dass auch hier die Ausgangswerte in bit-gespiegelter Reihenfolge vorliegen. Um diese in natürlicher Reihenfolge zu erhalten, müssen die Vektorindices als Zahlenwerte zur Basis 4 dargestellt und anschließend gespiegelt werden.
An den Knotenpunkten werden jeweils die Ergebnisse der Multiplikation aus Eingangswert und Twiddle-Faktoren addiert. Abbildung 3.6 zeigt das zugehörige Signalflussdiagramm.

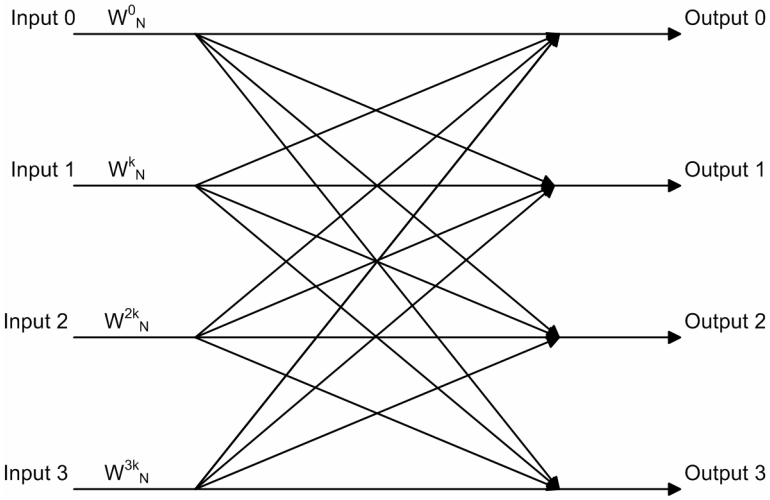


Abbildung 3.6 Radix 4 Butterfly

Allgemein gilt: zur Berechnung eines Radix 4 Butterflys sind 3 komplexe Multiplikationen und 12 komplexe Additionen notwendig, da $W_N^0 = 1$ ist.

Wenn die Additionen in zwei Schritten ausgeführt werden, lässt sich zeigen, dass die Anzahl der Additionen von 12 auf 8 komplexe Additionen reduziert werden kann [7].

Allgemein ergibt sich ein Rechenaufwand für eine Radix 4 FFT:

$3\frac{N}{4}\cdot\frac{\log_2 N}{2} = \frac{3}{8}N\cdot\log_2 N$ komplexe Multiplikationen

$8\frac{N}{4}\cdot\frac{\log_2 N}{2} = N\cdot\log_2 N$ komplexe Additionen.

Somit sind im Vergleich zur Radix 2 FFT 25% weniger komplexe Multiplikationen notwendig, wobei die Anzahl der Additionen unverändert ist.

Bei genauer Betrachtung der Twiddle-Faktoren lässt sich eine weitere Reduzierung der Multiplikationen durchführen. Für den Fall N=4 ergeben sich für die vier Twiddle-Faktoren Werte von $\pm 1, \pm j$. Bedenkt man nun, dass die Multiplikation mit $\pm j$ durch die Vertauschung von Imaginär- und Realteil erzeugt werden kann, ist für diesen Fall keine Multiplikation nötig (s. Abbildung 3.7).

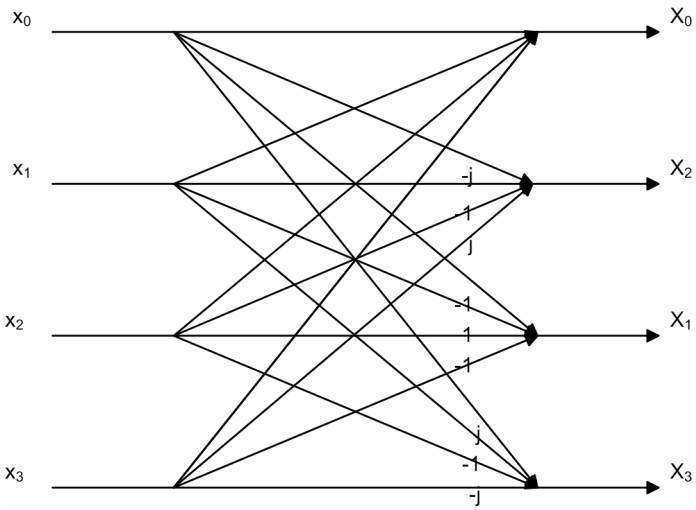


Abbildung 3.7 Beispiel zur Berechung der FFT ohne Multiplikation

Dieser im Vergleich zur Radix 2 FFT größere Zerlegungsaufwand ist nur für genügend große N sinnvoll. Für welche Anzahl N die Verwendung einer Radix 4 FFT sinnvoll ist, wird im Kapitel „Hardwarenahe Implementierung der FFT“ genauer untersucht.

3.6 Radix 8 FFT

Bei der Radix 8 FFT wird die Aufteilung der Eingangswerte, wie schon zuvor beschrieben, weiter fortgeführt. Zerlegt man die ursprüngliche DFT in acht einzelne Summen, die jeweils über ein Achtel der Gesamtlänge laufen, erhält man:

$$\begin{aligned} X_k = & \sum_{n=0}^{(N/8)-1} x_{8n} \cdot e^{-j8nk\frac{2\pi}{N}} + \sum_{n=0}^{(N/8)-1} x_{8n+1} \cdot e^{-j(8n+1)k\frac{2\pi}{N}} + \\ & \sum_{n=0}^{(N/8)-1} x_{8n+2} \cdot e^{-j(8n+2)k\frac{2\pi}{N}} + \sum_{n=0}^{(N/8)-1} x_{8n+3} \cdot e^{-j(8n+3)k\frac{2\pi}{N}} + \\ & \sum_{n=0}^{(N/8)-1} x_{8n+4} \cdot e^{-j(8n+4)k\frac{2\pi}{N}} + \sum_{n=0}^{(N/8)-1} x_{8n+5} \cdot e^{-j(8n+5)k\frac{2\pi}{N}} + \\ & \sum_{n=0}^{(N/8)-1} x_{8n+6} \cdot e^{-j(8n+6)k\frac{2\pi}{N}} + \sum_{n=0}^{(N/8)-1} x_{8n+7} \cdot e^{-j(8n+7)k\frac{2\pi}{N}} \end{aligned} \tag{3.18}$$

Mit $e^{-j82\pi/N} = W_{N/8}$ und durch Umstellen der Gleichung ergibt sich daraus:

$$\begin{aligned} X_k = & \sum_{n=0}^{(N/8)-1} x_{8n} \cdot W_{N/8}^{nk} + W_N^k \sum_{n=0}^{(N/8)-1} x_{8n+1} \cdot W_{N/8}^{nk} + \\ & W_N^{2k} \sum_{n=0}^{(N/8)-1} x_{8n+2} \cdot W_{N/8}^{nk} + W_N^{3k} \sum_{n=0}^{(N/8)-1} x_{8n+3} \cdot W_{N/8}^{nk} + \\ & W_N^{4k} \sum_{n=0}^{(N/8)-1} x_{8n+4} \cdot W_{N/8}^{nk} + W_N^{5k} \sum_{n=0}^{(N/8)-1} x_{8n+5} \cdot W_{N/8}^{nk} + \\ & W_N^{6k} \sum_{n=0}^{(N/8)-1} x_{8n+6} \cdot W_{N/8}^{nk} + W_N^{7k} \sum_{n=0}^{(N/8)-1} x_{8n+7} \cdot W_{N/8}^{nk} \end{aligned} \tag{3.19}$$

Um die Zerlegung nach Gleichung (3.18) durchführen zu können, muss die Anzahl der Eingangswerte einer Potenz der Basis 8 entsprechen.
Die Berechnung des Butterflys (s. Abbildung 3.8) erfolgt wie zuvor in Kapitel 3.4 beschrieben. Auch bei diesem Algorithmus zur Berechnung der FFT liegen die Ausgangswerte in bitgespiegelter Reihenfolge vor. Um diese in natürlicher Reihenfolge zu erhalten, müssen die Vektorindices als Zahlen zur Basis 8 dargestellt und anschließend gespiegelt werden. Abbildung 3.8 zeigt einen abgekürzten Radix 8 Butterfly.

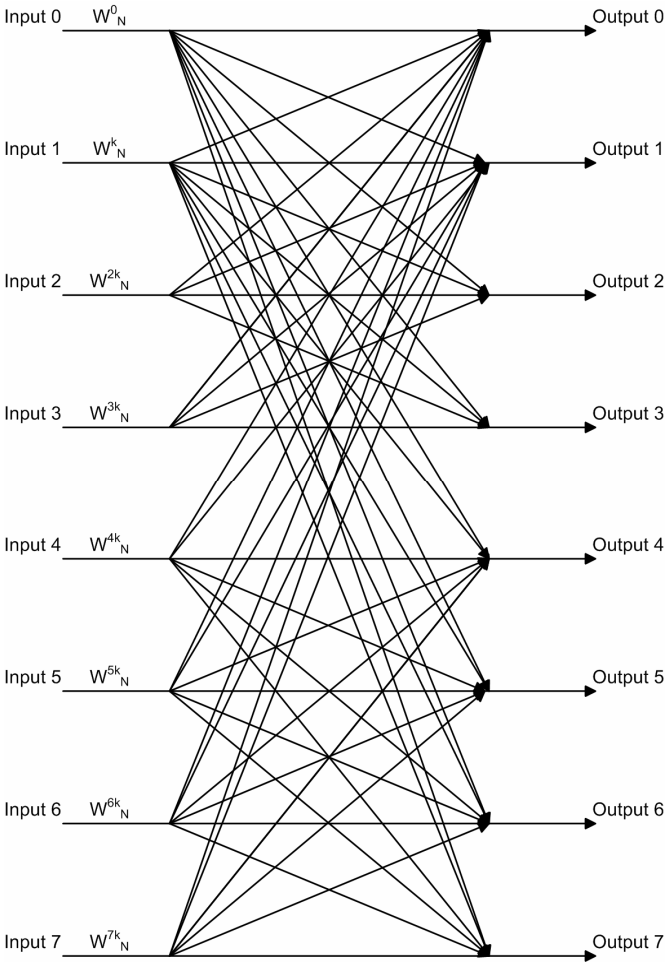


Abbildung 3.8 Abgekürzter Radix 8 Butterfly

In diesem Beispiel ergeben sich für die Twiddle-Faktoren die Werte $\pm 1, \pm j, \pm e^{j\pi/4}, \pm e^{-j\pi/4}$. Da die ersten zwei Faktoren keine Multiplikation erfordern und eine Multiplikation einer komplexen Zahl mit jedem der letzten zwei Faktoren nur je zwei reelle Multiplikationen benötigt, erfordert die Auswertung des Butterflys nur vier reelle Multiplikationen.

Allgemein ergibt sich ein Rechenaufwand für eine Radix 8 FFT:

$\frac{7}{24} N \cdot \log_2 N$ komplexe Multiplikationen

$N \cdot \log_2 N$ komplexe Additionen.

Somit ergeben sich theoretisch 41,7 % weniger komplexe Multiplikationen im Vergleich zur Radix2 FFT und 22 % weniger komplexe Multiplikationen bezogen auf die Radix 4 FFT. Die Anzahl der benötigten Additionen bleibt unverändert.

3.7 Alternative Algorithmen zur Berechnung der DFT

Neben den oben genannten Möglichkeiten gibt es noch weitere Formen zur effizienten Berechnung der DFT. Einige Algorithmen werden im Folgenden genauer erläutert.

3.7.1 FFT Algorithmen mit beliebigen Basen

Wie für obige Beispiele der Radix 2, 4, 8 FFT hergeleitet, lässt sich diese Zerlegung weiter fortführen. Dabei werden die zu berechnenden Butterflys bei jedem Schritt komplexer und damit die Programme zur Berechnung aufwändiger. Außerdem muss die Anzahl der Eingangswerte einer Potenz der jeweiligen Basis entsprechen. Somit können FFTs mit hohen Basen nur noch mit einer ganz bestimmten Anzahl von Eingangswerten durchgeführt werden.

3.7.2 Split-Radix FFT

Bei der Split Radix FFT handelt es sich um eine Kombination aus der Radix 2 und der Radix 4 FFT. Hierbei wird die Formel zur Berechnung der DFT

$$X_k = \sum_{n=0}^{N-1} x_n \cdot W_N^{kn} \qquad \textit{für } k = 0,1,2,...N-1 \tag{3.20}$$

wie schon bei der Radix 2 FFT in zwei Summen mit geraden und ungeraden Komponenten zerlegt, die jeweils über *N/2* laufen. Anschließend wird die zweite Summe wiederum in zwei Untersummen zerlegt, die jeweils über *N/4* der Werte aufsummieren. Durch diese Aufteilung ergibt sich:

$$X_k = \sum_{n=0}^{(N/2)-1} x_{2n} \cdot W_{N/2}^{kn} + W_N^k \sum_{n=0}^{(N/4)-1} x_{4n+1} \cdot W_{N/4}^{kn} + W_N^{3k} \sum_{n=0}^{(N/4)-1} x_{4n+3} \cdot W_{N/4}^{kn} \tag{3.21}$$

Eine weitere Zerlegung der Halben- und Viertelsummen der DFT führt zum gesamten Split-Radix Algorithmus. Damit ergibt sich der in Abbildung 3.9 [15] dargestellte Butterfly. Durch diese Art der Zerlegung entstehen L-förmige Strukturen, wodurch sich die Anzahl der Multiplikationen im Vergleich zur Radix 2 FFT um ein Drittel reduziert [9].

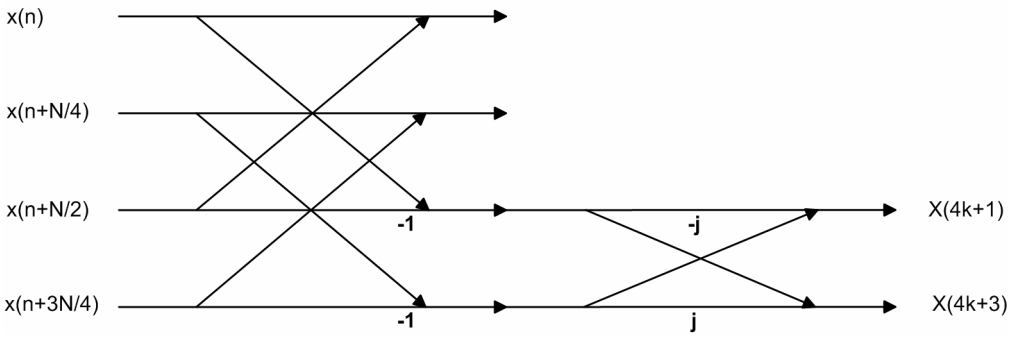


Abbildung 3.9 Split-Radix FFT Butterfly

Durch den im Vergleich zu anderen Algorithmen komplexen Aufbau lässt sich mit dem Split-Radix-Algorithmus die Hardware oft nicht optimal ausnutzen. Daher entsteht durch die Verwendung des Algorithmus in der Praxis kein wesentlicher Geschwindigkeitsvorteil gegenüber anderen optimierten Verfahren. Zum Beispiel sind für die Berechnung des Bruun´s Algorithmus nur *N*-2 mehr Rechenoperationen notwendig. Durch seine reguläre Struktur lässt dieser sich vor allem auf Hardwareplattformen mit mehreren Recheneinheiten effizienter umsetzen.

3.7.3 Primzahlen Algorithmus

Es kann gezeigt werden, dass es möglich ist, die Twiddle-Faktoren durch Substituieren und durch die Benutzung von relativen Primzahlen (d.h., diese besitzen keinen gemeinsamen Faktor) bei der Berechnung überflüssig zu machen und so Multiplikationen zu sparen. Für eine genaue Herleitung siehe [14].
Die Eliminierung der Multiplikationen mit Twiddle-Faktoren durch den Primfaktorenalgorithmus geht auf Kosten einer größeren Komplexität bei der Indexberechnung und Programmierung. Denn weder die Eingangs- noch die Ausgangswerte liegen in natürlicher Reihenfolge vor, sondern müssen vor und nach der Berechnung umsortiert werden.
Wo die Algorithmen nach Cooley und Tukey einen einfachen Butterfly besitzen, verlangt der Primzahlen Algorithmus für jeden Faktor verschiedene Butterflystrukturen. Aufgrund dieser Tatsache und da die Suche nach den Koeffizienten, welche die Bedingungen der relativen Primzahlen erfüllen, schwierig ist, wird dieser Algorithmus bei der praktischen Umsetzung nicht in Erwägung gezogen.

3.7.4 Weitere Lösungsverfahren

Neben den oben genannten gibt es noch weitere Algorithmen zur Lösung der DFT. Diese Verfahren beruhen auf der Berechnung der FFT unter Verwendung der Faltung. Zu diesen Algorithmen gehören der Winograd Algorithmus und der Chirp-Transformationsalgorithmus [14]. Da diese Verfahren sehr aufwändig zu implementieren sind und nur unter bestimmten

Voraussetzungen eine Ersparnis der Rechenzeit erlauben, werden sie hier nicht genauer betrachtet.
Ein weiteres Verfahren, mit dem nur bestimmte Spektralkomponenten eines abgetasteten Signals untersucht werden können, ist der Goertzel-Algorithmus. Er wird zur Erkennung von einzelnen Frequenzen (Tönen) in einem Signal eingesetzt. „Pro Berechnung einer Spektralkomponente sind bei dem Goertzel-Algorithmus 4*N* Additionen und 4*N* Multiplikationen notwendig. Vergleicht man diesen Aufwand mit dem Berechnungsaufwand bei der schnellen Fourier-Transformation ist der Goertzel- Algorithmus immer dann effizienter, wenn weniger als $5/6 \cdot \log_2(N)$ Spektralkomponenten berechnet werden sollen“ (Wikipedia Goertzel-Algorithmus, 02 01 2010)

3.8 2N Punkte Transformation mittels N Punkten FFT

Mit der Annahme, dass in der Praxis nur reellwertige Eingangssignale vorkommen, ergibt sich zur Lösung der FFT eine weitere Optimierungsmöglichkeit. In diesem Fall können die Symmetrieeigenschaften der DFT ausgenutzt werden, um eine effizientere Berechung durchzuführen.

Wenn x(t) ein komplexes zeitkontinuierliches Signal ist, dann lässt es sich wie folgt aufteilen:

$$x(t) = x_{re}(t) + j \cdot x_{im}(t) = x_{gerade}(t) + x_{ungerade}(t) \tag{3.22}$$

Für den Frequenzbereich gilt dann:

$$X(\omega) = X_{re}(\omega) + j \cdot X_{im}(\omega) \tag{3.23}$$

$$X_{re}(\omega) = \frac{1}{2}\left[X(\omega) + X^*(\omega)\right] \text{ und } X_{im}(\omega) = \frac{1}{2}\left[X(\omega) - X^*(\omega)\right] \tag{3.24}$$

wobei „*“ das konjugiert Komplexe der Funktion angibt.
Für den Fall, dass x(t) ein reelles Eingangssignal ist, d.h. $x(t)=x_{re}$ und $x_{im}=0$, kann gezeigt werden, dass

$$x_{gerade}(t) \circ\!\!-\!\!\bullet\, X_{re}(\omega) = \frac{1}{2}\left[X(\omega) + X^*(\omega)\right] \tag{3.25}$$

$$x_{ungerade}(t) \circ\!\!-\!\!\bullet\, jX_{im}(\omega) = j \cdot \frac{1}{2}\left[X(\omega) - X^*(\omega)\right] \tag{3.26}$$

Ähnliche Zusammenhänge ergeben sich für diskrete Signale. Es kann gezeigt werden, dass für ein reelles Eingangssignal x_{re} der Länge *N* gilt:

$$x_{re}(n) \circ\!-\!\bullet \frac{1}{2}\left[X(k) + X^*(<-k>_N)\right] \tag{3.27}$$

Und für ein imaginäres Signal $j \cdot x_{im}$ der Länge *N* gilt (wobei x_{im} selbst ein reelles Signal ist):

$$j \cdot x_{im}(n) \circ\!-\!\bullet \frac{1}{2}\left[X(k) - X^*(<-k>_N)\right] \tag{3.28}$$

wobei „$<-k>_N$“ die Rückwärtsoperation darstellt. Diese ist definiert als:

$$X(<-k>_N = \begin{cases} X(N-k) & \text{für } 1 \le k \le N-1 \\ X(k) & \text{für } k = 0 \end{cases} \tag{3.29}$$

Ähnliches gilt für die konjugiert-komplexe Darstellung:

$$X^*(<-k>_N = \begin{cases} X^*(N-k) & \text{für } 1 \le k \le N-1 \\ X^*(k) & \text{für } k = 0 \end{cases} \tag{3.30}$$

Wenn v(n) ein reelles Eingangssignal der Länge *2N* ist, lässt es sich in zwei Funktionen der Länge *N* wie folgt aufteilen:

$$\begin{aligned} &g(n) = v(2n), \\ &\qquad\qquad \text{für } 0 \le n \le N-1 \\ &h(n) = v(2n+1) \end{aligned} \tag{3.31}$$

wobei V(k), G(k) und H(k) die Fourier-Transformierten der jeweiligen Zeitbereichssignale darstellen. Wie in den Kapiteln zuvor gilt:

$$W_N = e^{-j2\pi/N} \tag{3.32}$$

Dann ergibt sich für die DFT V(k) von v(n)

$$\begin{aligned} V(k) &= \sum_{n=0}^{2N-1} v(n) W_{2N}^{nk} = \sum_{n=0}^{N-1} v(2n) W_{2N}^{2nk} + \sum_{n=0}^{N-1} v(2n+1) W_{2N}^{(2n+1)k} \\ &= \sum_{n=0}^{N-1} g(n) W_N^{nk} + \sum_{n=0}^{N-1} h(n) W_N^{nk} \cdot W_{2N}^{k} \\ &= \sum_{n=0}^{N-1} g(n) W_N^{nk} + W_{2N}^{k} \cdot \sum_{n=0}^{N-1} h(n) W_N^{nk} \end{aligned} \tag{3.33}$$

wobei hier der Zusammenhang aus Gleichung (3.32) genutzt wurde $W_{2N}^{2nk} = W_N^{nk}$.

Die erste Summe in der letzen Zeile von Gleichung (3.33) ist die Fourier-Transformierte G(k) von g(n), und die zweite Summe ist die Transformierte H(k) von h(n). Daher lässt sich V(k) auch wie folgt ausdrücken:

$$V(k) = G(< k >_N) + W_{2N}^{k} \cdot H(< k >_N), \qquad für 0 \le k \le 2N-1 \tag{3.34}$$

Aus Gleichung (3.27) ergibt sich für G(k):

$$G(k) = \frac{1}{2}\left[X(k) + X^*(< -k >_N)\right], \qquad für 0 \le k \le N-1 \tag{3.35}$$

und aus Gleichung (3.28) für H(k):

$$H(k) = \frac{1}{2j}\left[X(k) - X^*(< -k >_N)\right], \qquad für 0 \le k \le N-1 \tag{3.36}$$

Somit lässt sich der Imaginärteil der komplexen Zeitfunktion zur Berechnung der Transformation von *2N* reellen Abtastwerten mittels einer FFT für *N* komplexe Werte vorteilhaft ausnutzen.

In welchem Umfang dieser Zusammenhang eine Ersparnis der Rechenzeit ergibt, wird im Kapitel „Hardwarenahe Implementierung der FFT“ untersucht.

Des Weiteren wäre es denkbar, zwei *N* Punkte FFTs mittels einer FFT für *N* komplexe Werte durchzuführen. H(k) wäre dann die diskrete Transformation von h(n) und G(k) die diskrete Transformation von g(n) [16] und [18].

3.9 Fehlerbetrachtung und Vergleich der Algorithmen

In diesem Abschnitt werden die verschiedenen Verfahren zur Berechnung der DFT miteinander verglichen.

Wie allgemein bekannt, werden zur Berechnung der DFT N^2 komplexe Multiplikationen und $N \cdot (N-1)$ komplexe Additionen benötigt. Durch die Zerlegung in eine Radix 2 FFT und durch die Ausnutzung der Symmetrie der Twiddle-Faktoren reduziert sich der Rechenaufwand auf $N/2 \cdot \log_2 N$ komplexe Multiplikationen und $N \cdot \log_2 N$ komplexe Additionen. Dabei handelt es sich, wie auch bei den anderen Algorithmen, nicht um eine Näherungsrechnung, sondern um das identische Ergebnis. Die Reduzierung der Multiplikationen ergibt sich durch die Ausnutzung von Symmetrieeigenschaften der DFT. Durch die Verwendung des Radix 4 Algorithmus ergeben sich für die Berechnung $3/8 \cdot N \cdot \log_2 N$ komplexe Multiplikationen und $N \cdot \log_2 N$ komplexe Additionen.

Bei Verwendung des Radix 8 Algorithmus reduziert sich die Anzahl der Multiplikationen wiederum. Dies lässt sich theoretisch beliebig weiterführen, wobei die Reduktion der Multiplikationen immer weiter abnimmt und der Zerlegungsaufwand zur Berechnung zunimmt.

Allgemein ist es möglich, Algorithmen mit beliebigen Basen b, welche die Bedingung $b = 2^{\gamma}$ mit γ als positiver ganzer Zahl erfüllen, umzusetzen. Dabei muss die Anzahl der Eingangswerte *N* einer Potenz zur jeweiligen Basis b entsprechen. So ist z.B. eine Radix 8 (Basis 8) FFT nur mit $8^1 = 8,\ 8^2 = 64,\ 8^3 = 512,\ 8^4 = 4096,...$ Eingangswerten möglich. Für manche Anwendungen bedeutet dieser Zusammenhang eine große Einschränkung, da eventuell weniger Stützstellen benötigt werden. Somit werden durch die Verwendung dieses Algorithmus mehr Berechnungen ausgeführt als nötig und zusätzlisch Speicherplatz verschwendet. Im Vergleich zur direkten Berechnung bietet dieses Vorgehen jedoch eine große Ersparnis an Rechenoperationen. Die Anzahl, der zur Auswertung notwendigen Multiplikationen einer DFT und einer Radix 2 FFT ist in Abbildung 3.10 grafisch dargestellt.

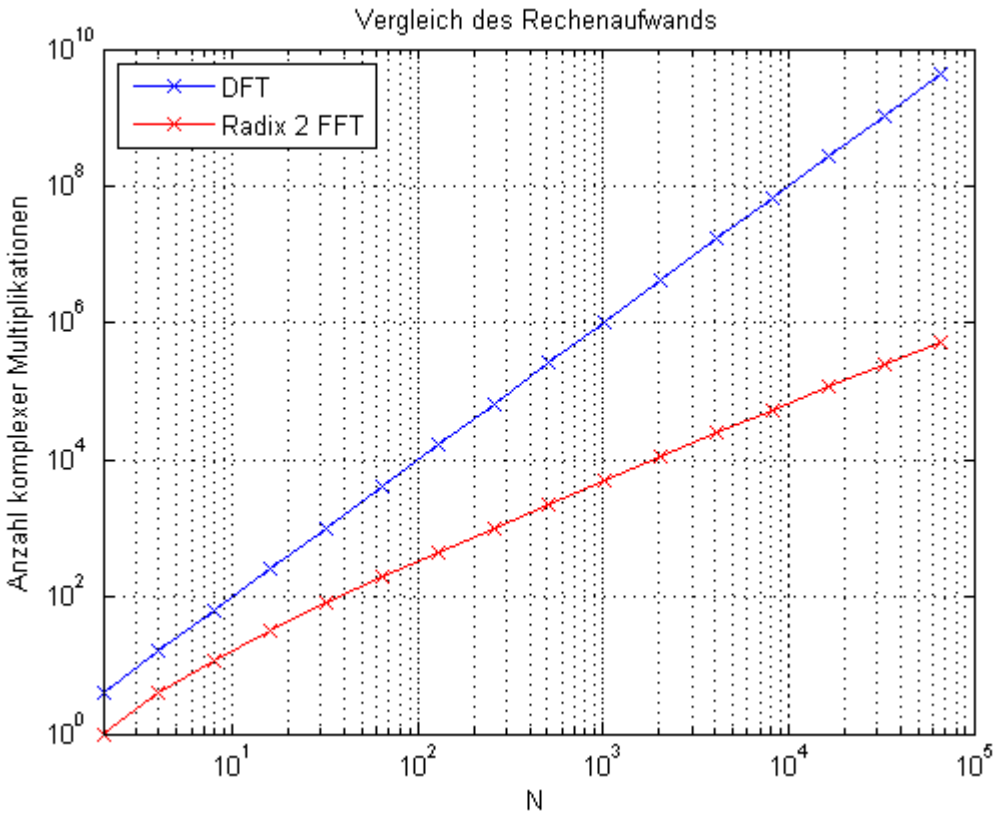


Abbildung 3.10 Vergleich des Rechenaufwands zwischen DFT und Radix 2 FFT

Die theoretische Anzahl an komplexen Multiplikationen von Radix 2, Radix 4 und Radix 8 FFT ist in Abbildung 3.11 dargestellt.

In der Grafik ist zu sehen, dass die Anzahl der Multiplikationen linear zunimmt, wobei die Ersparnis an Multiplikationen zwischen einer Radix 2 und Radix 8 FFT erst für große *N* signifikant wird.

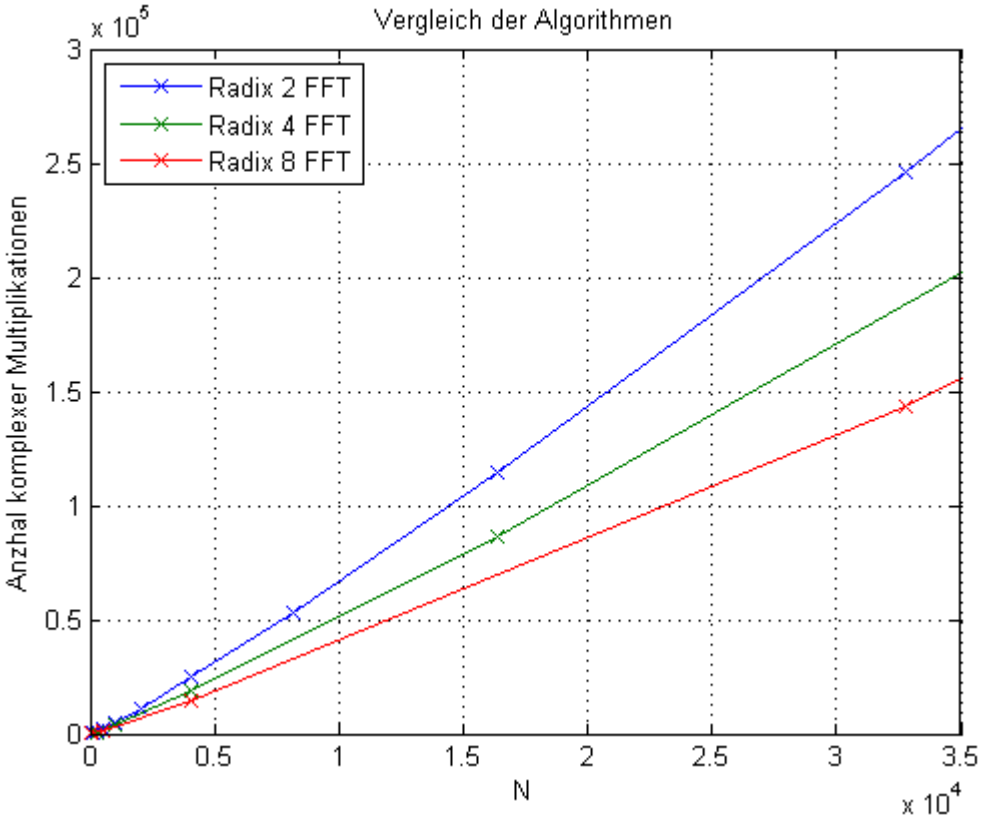


Abbildung 3.11 Vergleich der Radix n Algorithmen

In Tabelle 3.1 sind die theoretisch benötigten Rechenoperationen für die Berechnung einer 4096 Punkte FFT für die verschiedenen Basen dargestellt. Es ist zu sehen, dass die Anzahl der benötigten Additionen gleich bleibt und die Anzahl der Multiplikationen mit größeren Basen zwar abnimmt, jedoch die Ersparnis bei steigendem Zerlegungsaufwand, kleiner wird.

Tabelle 3.1 Rechenoperationen für den Radix 2, Radix 4 und Radix 8 FFT Algorithmus

Algorithmus	Anzahl der komplexen Multiplikationen	Anzahl der komplexen Additionen
Radix 2	24576	49152
Radix 4	18432	49152
Radix 8	14336	49152

Die Zerlegung der DFT hat neben der Ersparnis an Rechenzeit den Vorteil, dass die Berechnungen „in place“ durchgeführt werden. Wie im Signalflussgraphen in Abbildung 3.5 zu sehen ist, wird eine Menge von N Eingangswerten in eine andere Menge von N Werten durch die Auswertung der Butterflys berechnet. Dieser Vorgang wird $v = \log_2 N$ mal wiederholt und erzeugt die Berechnung der gewünschten FFT. Durch diese Struktur ist es möglich, die Eingangswerte einer Stufe mit dem Ergebnis dieser Stufe zu überschreiben, da die Eingangswerte für die weitere Berechnung nicht weiter benötigt werden.

3.9.1 Quantisierungseffekt Fix- und Floating-Point Berechnung

Wie in den Signalflussgrafen (z.B. Abbildung 3.5) zu sehen ist, bedeutet ein Übergang von jedem Knoten zu jedem anderen Knoten eine Multiplikation einer komplexen Konstanten mit der Betragsamplitude Eins (da jede Übertragungsfunktion eines Astes entweder eins oder eine ganze Potenz von W_N ist). Außerdem wird eine komplexe Addition zur Auswertung benötigt.
Bei einer Implementierung in Fixpoint–Arithmetik entsteht bei jeder Multiplikation ein Rundungsfehler, welcher auch als Rundungsrauschen aufgefasst werden kann.
Außerdem müssen bei dieser Art der Realisierung, Überläufe des Wertebereichs vermieden werden. Wenn die Ausgangswerte der FFT kleiner als eins sind, muss die Amplitude an den Punkten in jedem Array kleiner als eins sein, d.h. es kann zu keinem Überlauf kommen. Um dies zu erreichen, müssen die Eingangswerte $x[n]$ die Bedingung

$$|x[n]| \le \frac{1}{N} \qquad \textit{für} \qquad 0 \le n \le N-1 \tag{3.37}$$

erfüllen. Somit ist sichergestellt, dass

$$|X[k]| \le 1 \qquad \textit{für} \qquad 0 \le k \le N-1 \tag{3.38}$$

ist und es in keiner der Stufen zu einem Überlauf kommen kann.
Durch eine solche Skalierung werden die Eingangswerte sehr klein, dies führt dazu, dass die Rundungsfehler bei der Berechnung einen großen Einfluss auf das Endergebnis haben. Das Signal-Rauschverhältnis (engl. Signal to Noise Ratio) ist in diesem Fall proportional zu N^2 [14]. Da die maximale Amplitude in jeder Stufe nicht mehr als den Faktor 2 größer werden kann, kann ein Überlauf vermieden werden, indem sichergestellt wird, dass $|x[n]| < 1$ gilt und vor dem Eingang jeder Stufe das Signal um den Faktor 2 abgeschwächt wird. In diesem Fall ist die Amplitude des Ausgangssignals um den Faktor *1/N* kleiner als bei der ursprünglichen DFT (Gleichung (3.1)).
Obwohl der quadratische Mittelwert des Ausgangssignals nur das $1/N$ –fache beträgt, was ohne eine Skalierung zu erreichen wäre, kann die Eingangsamplitude nun N –fach größer gewählt werden, ohne dass ein Überlauf auftreten kann. Durch diese Art der Skalierung wird das Rundungsrauschen der ersten Stufen zusätzlich abgeschwächt. Somit fällt der Rundungsfehler am Ende der Berechnung kleiner aus als bei der zuvor betrachteten Skalierung. Das Signal-Rauschverhältnis ist in diesem Fall proportional zu N und nicht mehr zu N^2 [14].
Ein großer Teil des auftretenden Rauschens ist auf die Abnahme des Signalpegels zurückzuführen, welche sich durch eine Skalierung mit dem Faktor 2 ergibt.

Eine Möglichkeit zur Reduzierung der Rundungsfehler ist die Verwendung von Floating-Point Datentypen, da es bei diesen Datentypen aufgrund des großen darstellbaren Wertebereiches faktisch nicht zu Überläufen kommen kann. Damit kann eine Skalierung und der damit verbundene Fehler vermieden werden. „Die Auswirkungen der Rundung bei Gleitkommaarithmetik [Floating-Point Arithmetik, Anm. d. Verf.] bei der FFT wurden sowohl theoretisch als auch experimentell von Gentleman und Sande (1966) … analysiert“ (Oppenheim 1992, Seite 728). Diese Untersuchungen haben ergeben, dass die Verschlechterung des Signal-Rauschverhältnisses bei großen N einen wesentlich geringeren Einfluss hat als dies bei der Implementierung in Fixpoint Arithmetik der Fall ist.

3.9.2 Koeffizientenquantisierung bei der FFT

Wie auch bei anderen Berechnungen findet bei der FFT eine Quantisierung der Koeffizienten statt. Da die Koeffizientenquantisierung schwer zu berechnen ist, wurden nützliche Ergebnisse durch statistische Analyse von Oppenheim und Weinstein ermittelt [14]. Dafür wurde zu jedem Koeffizienten eine Folge weißen Rauschens addiert.
Obwohl der Fehler durch Koeffizientenquantisierung nicht genau berechnet werden kann, liefert die in Abbildung 3.12 [14] gezeigte Analyse von Oppenheim und Weinstein eine Abschätzung des Fehlers. In der Abbildung ist das Verhältnis des mittleren quadratischen Amplitudenfehlers und des mittleren Quadrats des Ausgangssignals in Abhängigkeit von $v = \log_2 N$ dargestellt.

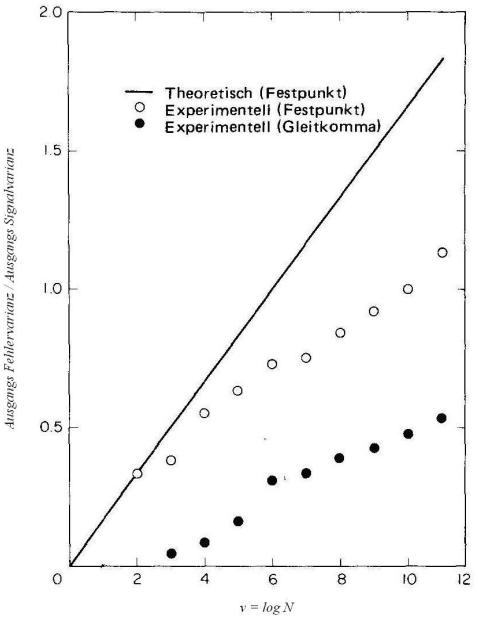


Abbildung 3.12 Koeffizientenquantisierungsfehler

4 Hardwarenahe Implementierung der FFT

In diesem Kapitel werden die zuvor diskutierten Ansätze zur Berechnung der DFT auf dem Signalprozessor „TMS320C6713“ der Firma *Texas Instruments* in der Programmiersprache C umgesetzt und in mehreren Schritten optimiert. Dabei werden die Geschwindigkeiten der Algorithmen (Radix 2 und Radix 4) sowie die Implementierung in Fixpoint und Floating-Point Datentypen miteinander verglichen.
Es wird nicht nur versucht, die Rechenzeit des Algorithmus selbst zu verkürzen, sondern auch das Vor- und Nachbereiten der Daten. Dazu gehören das Einlesen der Abtastwerte sowie die Typumwandlung („cast“) auf einen anderen Datentyp und die Berechnung des Ergebnisses in natürlicher Reihenfolge.

4.1 Struktur der Softwareentwicklung

Wie zuvor beschrieben erfolgt die Entwicklung der Software zur Berechnung der diskreten Fourier-Transformation in mehreren Schritten. Zu Beginn werden anhand einer Radix 2 FFT (DIT) die Unterschiede bei der Umsetzung in Floating-Point und Fixpoint -Darstellung untersucht. Dies findet in mehreren Optimierungsstufen statt. Zunächst wird der C Programmcode möglichst effizient programmiert. Anschließend wird der Einfluss der Compiler Optimierung untersucht.
Daraufhin werden die Radix 2 und Radix 4 Algorithmen implementiert und die einzelnen Programmabschnitte wiederum in mehreren Stufen optimiert. Dazu gehören eine effiziente Gestaltung des C-Codes, Intrinsic-Funktionen, Compiler Optimierung und die Umsetzung rechenaufwändiger Programmabschnitte in Assembler-Code.

4.2 Umsetzung der Algorithmen in C-Code

Zu Beginn wird ein Programm für eine Radix 2 FFT (DIT) unter Verwendung des Datentyps „float“ geschrieben. Wie in Kapitel 2.2 beschrieben unterstützt der Prozessor sowohl Floating-Point als auch Fixpoint -Operationen. Die Fließkomma-Darstellung hat den Vorteil, dass es aufgrund des großen darstellbaren Wertebereiches faktisch nicht zu Überläufen kommen kann. Ein weiterer Vorteil ist die höhere Genauigkeit gegenüber Festkomma-Datentypen. Dadurch entstehen geringere Rundungsfehler. Ein Nachteil dieser Art der Zahlendarstellung ist, dass der Prozessor für die Berechnung einer Multiplikation und einer Addition jeweils vier Taktzyklen benötigt. Zum Vergleich: zur Berechnung einer 16 Bit Festkomma-Multiplikation benötigt der Prozessor zwei Taktzyklen und zur Berechnung einer Addition lediglich einen Takt.

4.2.1 Floating-Point Implementierung

Das Programm „Radix_2_Float“ liest die Daten in der Interrupt Routine vom AIC23 Codec ein. Dieser liefert bei jedem Abtastzeitpunkt für jeden der beiden Kanäle einen 16 Bit Wert vom Datentyp „short“. Die Anzahl der Werte N wird am Anfang des Programms mit einer Präprozessoranweisung festgelegt (N muss dabei einer Potenz zur Basis zwei entsprechen).
In der Interrupt Routine (s. Abbildung 4.1) werden wahlweise die Daten des linken oder des rechten Kanals in das Array „in_buf“ der Länge N gespeichert. Nachdem N Abtastwerte aufgenommen wurden, werden diese im Hauptprogramm weiter verarbeitet.

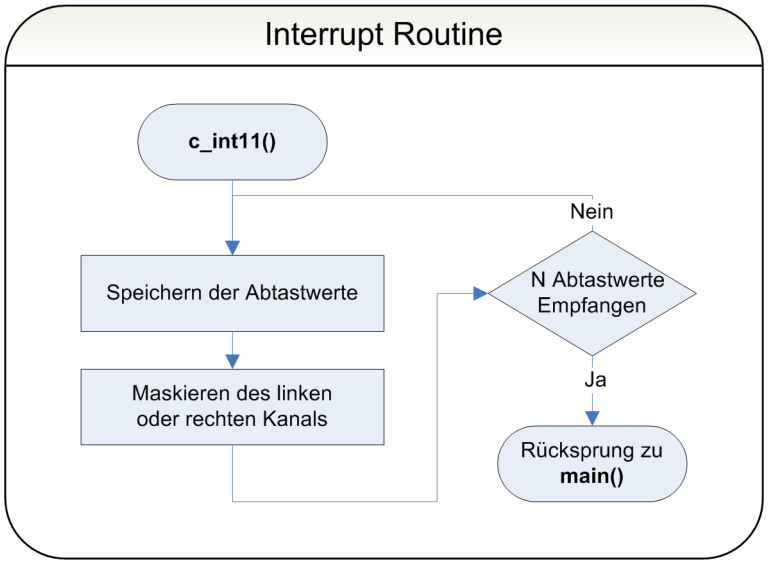


Abbildung 4.1 Interrupt Routine zum Abtasten der Eingangswerte

Nach dem Start des Programms werden zunächst alle zur Berechnung nötigen Variablen initialisiert. Anschließend wird mit dem Funktionsaufruf „comm_intr( )“ die Kommunikation über den „McBSP“ initialisiert und der „AIC23 Codec“ zum Abtasten der Eingangswerte konfiguriert und der Interrupt „c_int11“ aktiviert. Im nächsten Schritt werden die nötigen Twiddle-Faktoren berechnet und in einem Array vom Datentyp „float“ gespeichert. Daraufhin wartet das Programm, bis N neue Abtastwerte zur Verfügung stehen. Diese Daten werden auf den Datentyp „float“ konvertiert (gecastet). Anschließend werden sie zur Unterdrückung des Leck-Effekts[4] (engl. Leakage Effect) zunächst mit einem Hamming-Fenster gewichtet und dann in bit-gespiegelter Reihenfolge in einem Array gespeichert. Dieses Array hat die Länge $2 \cdot N$, da in ihm die Eingangsfolge als komplexe Werte gespeichert wird (Imaginärteil beträgt 0).

[4] Fehler, der bei der Berechnung durch die endliche Anzahl an berechneten Abtastwerten entsteht. Für genauere Information siehe [15] Seite 489

Anschließend wird das Array zusammen mit den zuvor berechneten Twiddle-Faktoren zur Berechnung der FFT an die Funktion „radix2( )“ übergeben (s. Abbildung 4.2). Diese Funktion berechnet mit Hilfe der Unterfunktion „butterfly( )“ die FFT.

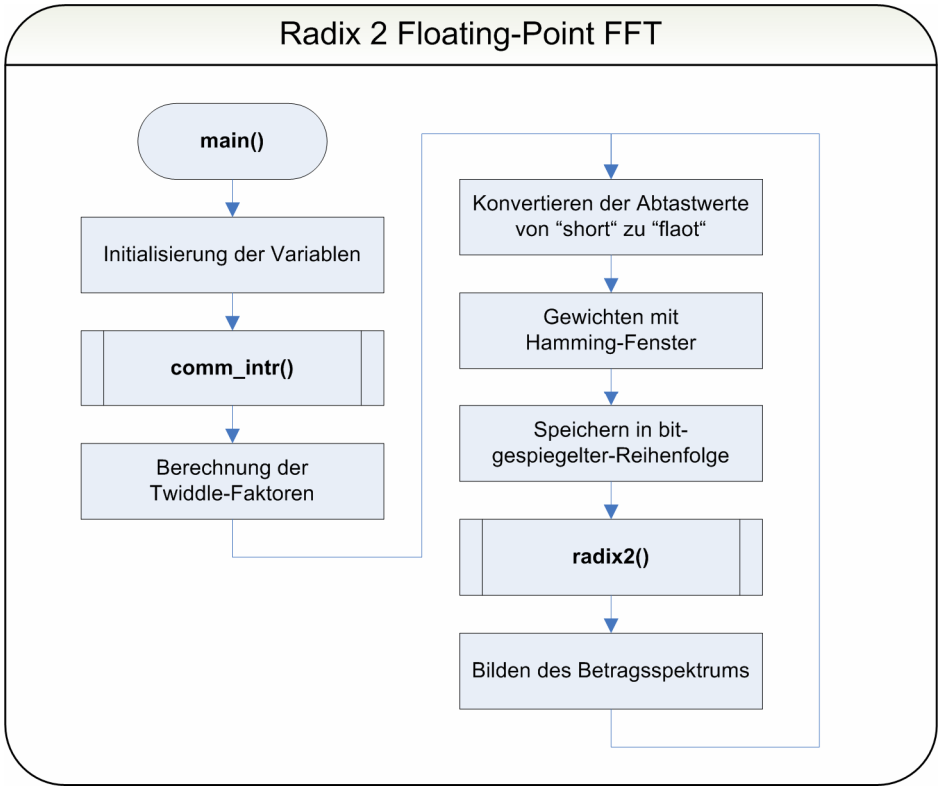


Abbildung 4.2 Programmablauf Radix 2 FFT mit Flaoting-Point Datentypen

Dazu werden alle in dieser Funktion benötigten Variablen deklariert und anschließend wird jeweils ein Pointer auf den oberen und den unteren reellen Eingangswert eines Butterflys gesetzt. Diese beiden Pointer werden mit den dazugehörigen Twiddle-Faktoren zur Berechnung an die Funktion „butterfly( )“ übergeben.
Dieser Vorgang wird solange wiederholt, bis alle Butterflys eines Blocks, dann alle Blöcke einer Stufe und zum Schluss alle Stufen berechnet wurden (s. Abbildung 4.3). Dann ist die Funktion beendet und im Hauptprogramm wird abschließend das Betragsspektrum gebildet. Daraufhin wartet das Programm, bis N neue Abtastwerte von der Interrupt Routine gespeichert wurden. Die Eingangswerte werden in bit-gespiegelter Reihenfolge bereitgestellt (die Indexvariable zur Umsortierung der Bits wurde zuvor in Matlab berechnet und in einem „Headerfile“ bereitgestellt), und somit stehen nach jedem Schleifendurchlauf die Ergebnisse in natürlicher Reihenfolge in der Variablen „out_buf“ zur Weiterverarbeitung zur Verfügung.

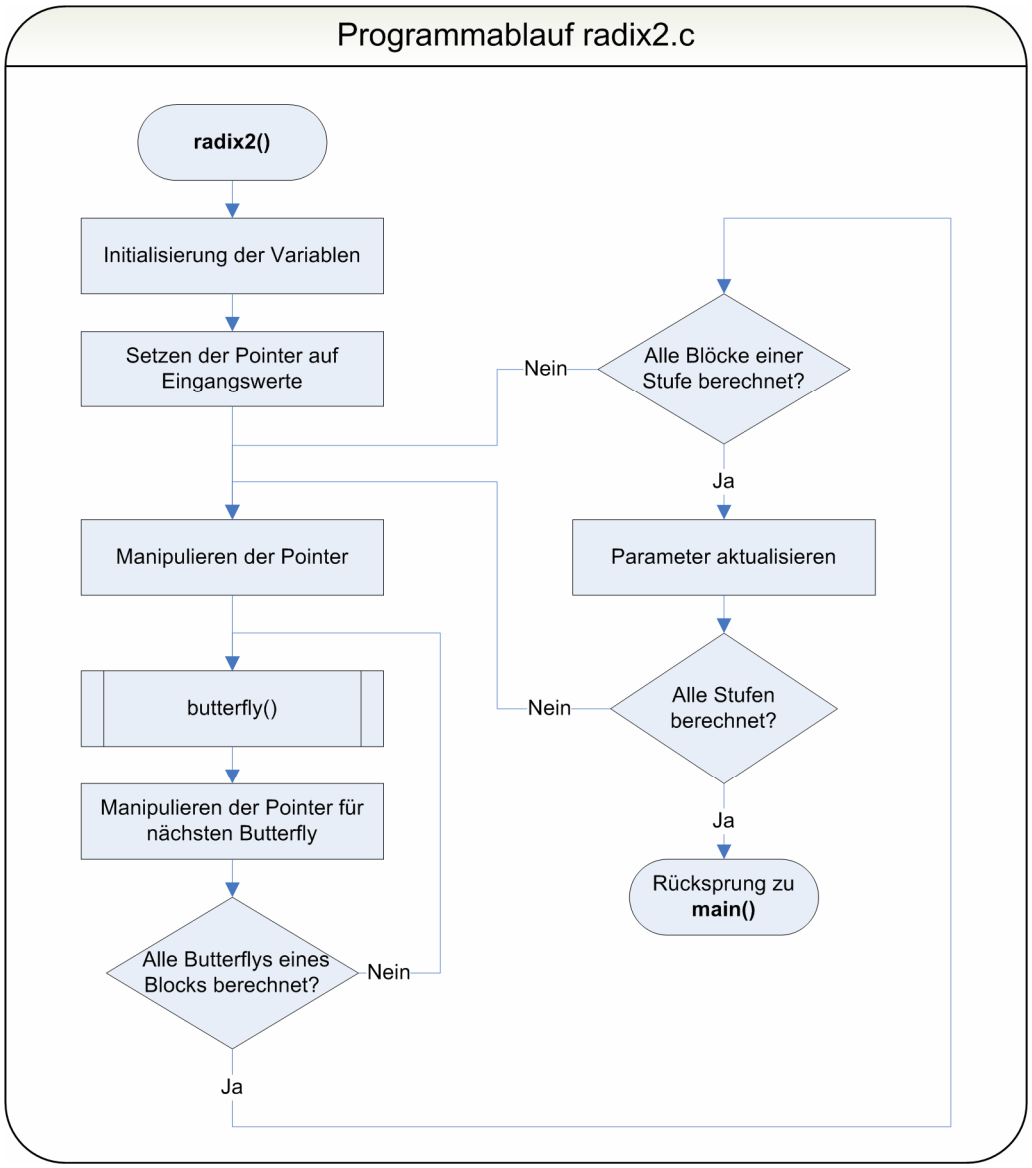


Abbildung 4.3 Funktionsaufruf radix2.c

An die Funktion „butterfly( )“ (s. Abbildung 4.4) werden zwei Pointer auf die benötigten Eingangswerte und die dazugehörigen Twiddle-Faktoren übergeben. Damit wird der obere und untere Knoten des Butterflys nach Real- und Imaginärteil berechnet und das Ergebnis „in place“ zurück geschrieben. Somit befindet sich das Ergebnis der Berechnung im alten Eingangsvektor.

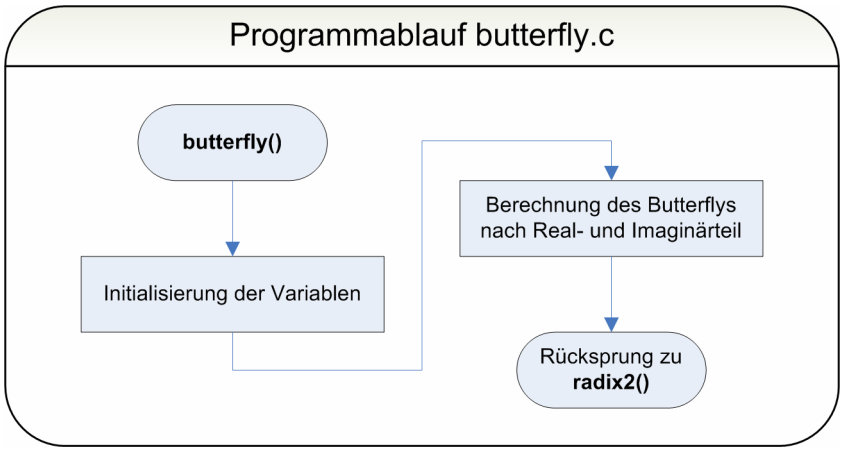


Abbildung 4.4 Funktionsaufruf butterfly.c

4.2.2 Fixpoint Implementierung

Um die Geschwindigkeit der Berechnung zu erhöhen, wird das vorhandene Programm so umgeschrieben, dass ausschließlich Festkomma-Datentypen verwendet werden. Bei der Umsetzung ist zu beachten, dass es nicht zu Überläufen kommt und die Rundungsfehler möglichst gering ausfallen.

Das Speichern der abgetasteten Werte findet wie im vorherigen Programm in der Interrupt-Routine statt. Als nächstes werden die Daten wiederum mit einem Hamming-Fenster gewichtet und anschließend in bit-gespiegelter Reihenfolge gespeichert (s. Abbildung 4.5). Bei dieser Art der Implementierung findet keine Typumwandlung (cast) statt. Somit ergibt sich neben der Ersparnis an Rechenzeit bei den Additionen und Multiplikationen noch eine weitere Zeitersparnis.

Das manipulierte Array (bit reversed) wird wiederum mit den zuvor berechneten Twiddle-Faktoren vom Datentyp „short“ an die Funktion „radix2.c“ übergeben. Bei der Berechnung der einzelnen Butterflys findet nach jeder Multiplikation eine Division mit dem Faktor zwei statt, um einen Überlauf zu vermeiden. Diese Operation wird zeitsparend durch eine Bitschiebeoperation um ein Bit nach rechts realisiert.

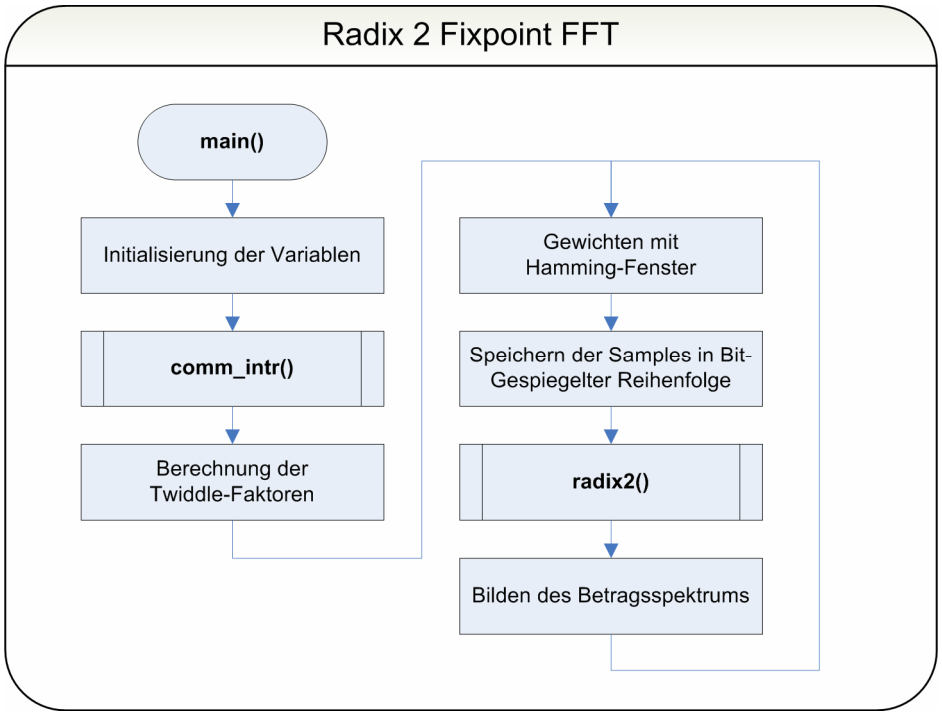


Abbildung 4.5 Programmablauf Radix 2 FFT mit Fixpoint Datentypen

Wenn bei der Weiterverarbeitung des Ergebnisses die Höhe der Amplitude entscheidend ist, muss berücksichtigt werden, dass jedes Ergebnis bei der Berechnung des Butterflys mit dem Faktor zwei skaliert wird. Die Lage der Spektrallinien wird durch die Skalierung jedoch nicht beeinflusst.

Auch in diesem Programm wird die Berechnung „in place“ durchgeführt, so dass nach der Bildung des Betragsspektrums das Ergebnis im Vektor „out_buf“ vom Typ „integer“ zur Verfügung steht.

Für die Funktionen „radix2.c“ und „butterfly.c“ werden angepasste Programmcodes aus dem Praktikum „Digitale Signalverarbeitung“ verwendet.

4.2.3 Vergleich der Ergebnisse

Die beiden unterschiedlichen Arten der Umsetzung werden mit verschiedenen Eingangssignalen bei unterschiedlichen Frequenzen getestet. Bei der Anregung mit einem 1 kHz Rechtecksignal ergeben sich die in Abbildung 4.6 dargestellten Spektrallinien der FFT in Floating-Point-Darstellung und in Abbildung 4.7 in Fixpoint-Darstellung.

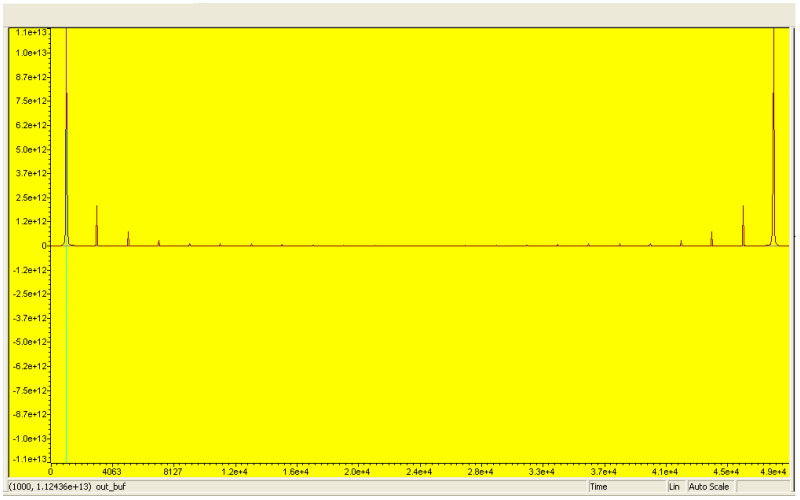


Abbildung 4.6 Ergebnis der FFT in Floating-Point

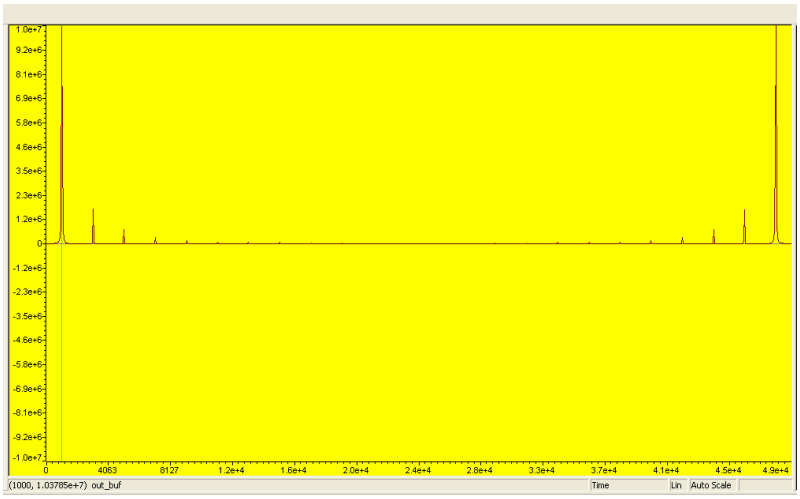


Abbildung 4.7 Ergebnis der FFT in Fixpoint

In den Abbildungen lässt sich, abgesehen von der Höhe der Amplituden, kein Unterschied erkennen. Zur Evaluierung der beiden Ergebnisse wird nach der Berechnung die inverse Fourier-Transformation durchgeführt. Durch grafische Analyse lässt sich bei den beiden Ausgangssignalen kein Unterschied zum Eingangssignal feststellen.
Zusätzlich wurden die berechneten Werte der FFT zur Kontrolle über den „Line Out“-Ausgang des Bords ausgegeben und auf einem Oszilloskop angezeigt. In Abbildung 4.8 sind die erwarteten Spektrallinien bei der Grundschwingung und den ungradzahligen Vielfachen der Grundschwingung zu sehen.

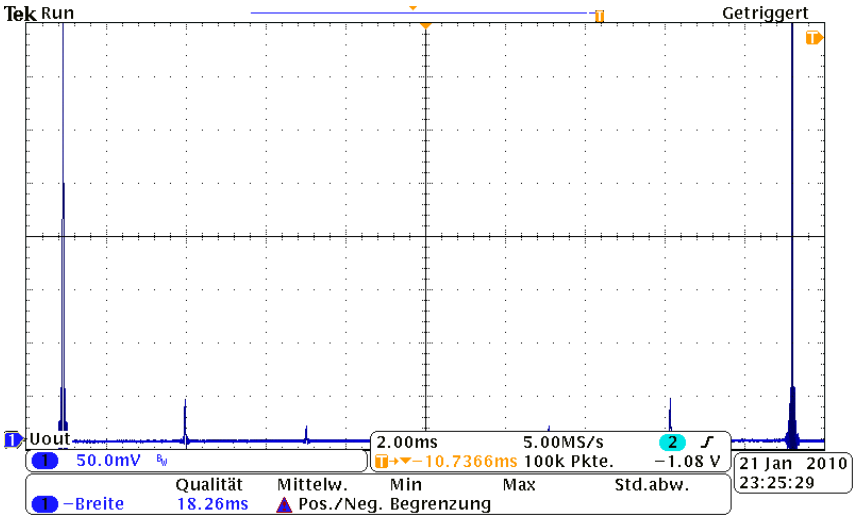


Abbildung 4.8 Spektrallinien 1kHz Rechteck auf einem Oszilloskop

Um die Algorithmen bezüglich ihrer Geschwindigkeit vergleichen zu können, werden die Zeiten für die Berechnung bei einer unterschiedlichen Anzahl von Eingangswerten gemessen. Zum einen wird mit Hilfe des CCS-Simulators und des „Profilers“ die Anzahl der benötigten Takte für eine Berechnung der FFT (ein Schleifendurchlauf) inklusive der Funktionsaufrufe gemessen und anhand der Taktfrequenz von 225 MHz in eine Ausführungszeit umgerechnet. Zum anderen werden die Zeiten zusätzlich auf der Hardwareplattform mit Hilfe eines Oszilloskops gemessen. Dazu wird ein Pin des Controllers zu Beginn der Berechnung eingeschaltet und nach Beendigung der Berechnung ausgeschaltet. Die Schaltzeiten, die der Controller benötigt, um den Pin zu setzen bzw. rückzusetzen, können dabei vernachlässigt werden, da diese im Vergleich zur Programmlaufzeit sehr klein sind. Außerdem müssen alle Interrupts für die Zeitmessung deaktiviert werden, da diese die Programmlaufzeit beeinflussen.
Wie in Tabelle 4.1 zu sehen ist, liegt die mittels „Profiler“ ermittelte Zeit bei jedem Messpunkt etwa um den Faktor 0,95 niedriger als der gemessene Wert. Um die beiden Programme unabhängig von äußeren Einflüssen vergleichen zu können, werden für die weiteren Betrachtungen die im Simulator ermittelten Werte verwendet.

Tabelle 4.1 Laufzeit der Floating-Point und Fixpoint Implementierung

	Floating-Point			Fixpoint		
N	Cycles incl.Total	Theo. Zeit[ms]	Zeit [ms]	Cycles incl.Total	Theo. Zeit[ms]	Zeit [ms]
64	51192	0.23	0.25	45923	0.20	0.22
128	115497	0.51	0.55	103241	0.46	0.47
256	257849	1.15	1.21	229691	1.02	1.07
512	571392	2.54	2.68	506394	2.25	2.36
1024	1266749	5.63	5.88	1110702	4.94	5.16
2048	2787329	12.39	12.9	2438669	10.84	11.3
4096	6033829	26.82	28	5348418	23.77	24.8

In Abbildung 4.9 sind die Messwerte aus Tabelle 4.1 dargestellt. Wie zu sehen ist, benötigt die Fixpoint FFT für die Berechnung weniger Zeit als die Floating-Point Implementierung. Dies ist mit der Dauer der Fix- bzw. Floating-Point Operationen zu begründen (4 Takte für eine Floating-Point Multiplikation und 2 Takte für eine Fixpoint Multiplikation).

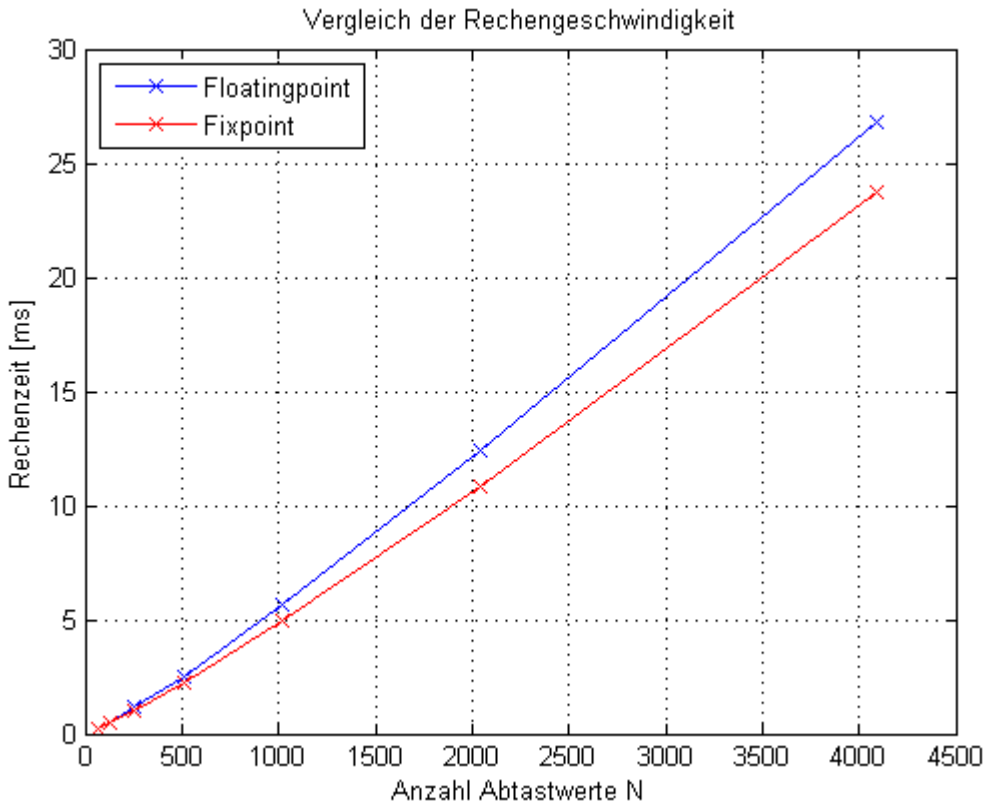


Abbildung 4.9 Vergleich der Floating-Point und Fixpoint Implementierung

Tabelle 4.2 zeigt die Laufzeiten der gleichen Programme wie zuvor, jedoch durch den CCS-Compiler „o3“ optimiert. Es ist zu sehen, dass sich die Ausführungszeiten für die Berechnung der FFT wesentlich verkürzen. Zum Beispiel hat die „Fixpoint FFT“ ohne Optimierung für 1024 Punkte 4,94 ms benötigt im Vergleich zu 1,6 ms mit Compiler-Optimierung.
Des Weiteren ist auffällig, dass der Zeitunterschied zwischen den beiden Arten der Implementierung kleiner ist als vor der Compiler-Optimierung.
Der Grund dafür ist, dass der Compiler ein so genantes „Pipelining“ des Programmcodes durchführt. Dabei wird ausgenutzt, dass z.B. eine Floating-Point Multiplikation nur einen Takt plus drei Takte Verzögerung benötigt, bis das Ergebnis der Multiplikation zur Verfügung steht. Während dieser Verzögerung werden neue Multiplikationen ausgeführt. So ist es theoretisch möglich, dass in jedem Takt von jeder Multiplikationseinheit (.M1 und .M2) eine Multiplikation durchgeführt wird. Durch diese Art der Programmstruktur benötigt der Prozessor effektiv weniger als vier Takte für eine Floating-Point Multiplikation. Dadurch verringert sich der Geschwindigkeitsvorteil der Fixpoint FFT.

Tabelle 4.2 Laufzeit der Floating-Point und Fixpoint Implementierung o3 optimiert

	Floatingpoint			Fixpoint		
N	Cycles	Theo. Zeit[ms]	Zeit [ms]	Cycles	Theo. Zeit[ms]	Zeit [ms]
64	16951	0.08	0.1	14502	0.06	0.08
128	38520	0.17	0.21	32772	0.15	0.18
256	86673	0.39	0.46	73366	0.33	0.4
512	193879	0.86	1.02	162790	0.72	0.86
1024	440277	1.96	2.44	359738	1.60	1.88
2048	993572	4.42	5.1	811079	3.60	4.18
4096	2163531	9.62	11.1	1839605	8.18	9.48

In Abbildung 4.10 sind die Laufzeiten der beiden Programme nach der Compiler-Optimierung dargestellt. Durch die Optimierung sind beide Programme wesentlich schneller geworden. Die verbleibende Zeitdifferenz zwischen den beiden Varianten ist auf die Konvertierung der Daten von „short“ zu „float“ zurückzuführen, die bei der Floating-Point Implementierung vorzunehmen ist. Diese Konvertierung benötigt z.B. für 1024 Werte 25µs. Somit würde eine Fixpoint Implementierung, nur auf die eigentliche Rechenzeit bezogen, keine wesentliche Zeitersparnis bedeuten.

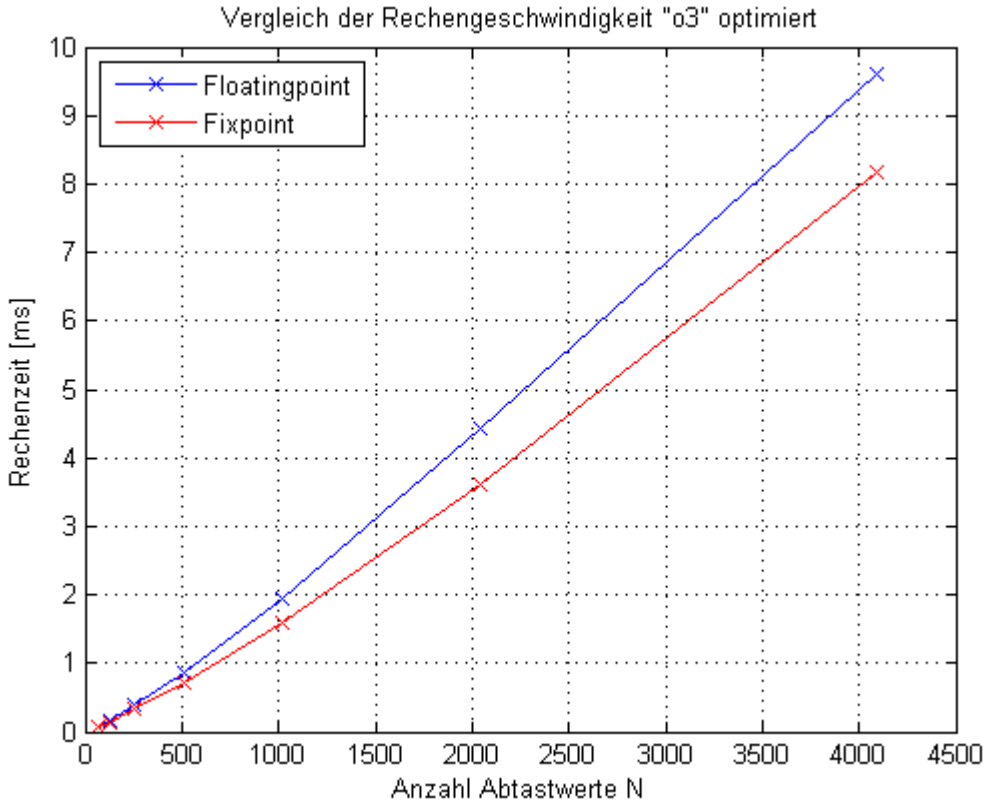


Abbildung 4.10 Vergleich der Floating-Point und Fixpoint Implementierung o3 optimiert

4.3 Hardware-Optimierung

Bei der Aufnahme der Abtastwerte wird der Programmablauf durch einen Interrupt unterbrochen, um die Abtastwerte äquidistant aufzunehmen. Somit wird die Berechnung der FFT zur Speicherung der Abtastwerte unterbrochen und nach Beendigung der Interrupt Routine fortgesetzt. Diese Art der Speicherung ist nachteilig, da der Programmablauf für jeden einzelnen Abtastwert unterbrochen werden muss.
Der EDMA Controller des Prozessors bietet die Möglichkeit, die abgetasteten Werte zu Speichern, ohne die CPU zu nutzen. Somit steht diese ausschließlich für die Berechnungen zur Verfügung.

4.3.1 EDMA Konfiguration

Um beim Aufnehmen der Abtastwerte die CPU nicht zu unterbrechen, wird der EDMA so konfiguriert, dass er die eingestellte Anzahl Abtastwerte ohne Einfluss der CPU speichert und nach Beendigung des Vorgangs einmalig einen Interrupt anfordert. Anschließend beginnt der EDMA Controller erneut mit dem Aufnehmen der nächsten *N* Abtastwerte und die gespeicherten Werte stehen der CPU währenddessen zur Berechnung zur Verfügung.
Dazu wird im Options-Register (OPT) (s. Abbildung 4.11) die Größe des zu empfangenen Datentyps auf 32 Bit (linker und rechter Kanal) und die Speicherung der Daten in ein eindimensionales Array festgelegt. Des Weiteren wird der EDMA so konfiguriert, dass er nach der Aufnahme der *N* Abtastwerte einen Interrupt anfordert, um den Datensatz an die CPU zu übergeben.
Im Source Adress Register wird der „McBSP1“-Kanal als Empfangsadresse eingestellt. Im Count Register wird die Anzahl der aufzunehmenden Abtastwerte (Anzahl Elemente im 1D Array) auf *N* Abtastwerte festgelegt. Als Zieladresse (destination adress) wird die Adresse des Arrays „in_buf“ angegeben, in dem die Werte gespeichert werden sollen.
Nach der Konfiguration wird der EDMA gestartet und beginnt mit dem Aufnehmen und Speichern der Abtastwerte. Um kontinuierlich Abtastwerte aufzunehmen, muss der EDMA-Handler auf sich selbst „verlinkt“ werden. Dazu wird im Register „link adress“ seine eigene Adresse eingetragen. Somit ruft dieser sich selbst wieder auf.

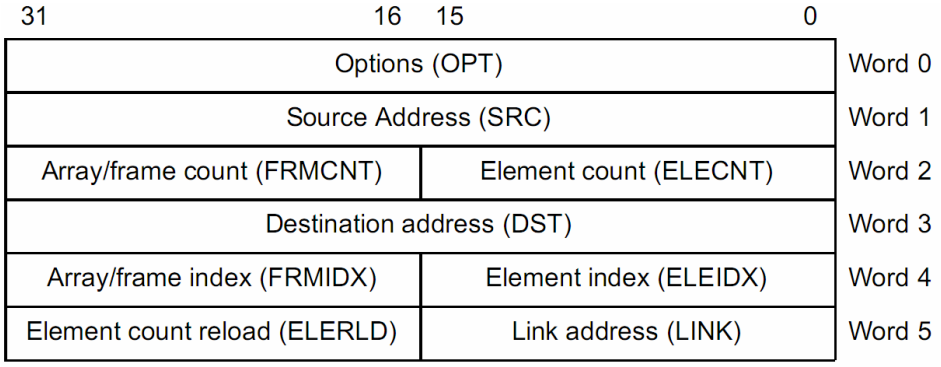


Abbildung 4.11 Aufbau des EDMA Parameter RAM

4.3.2 Ping Pong Buffering

Unter „Ping-Pong“ Buffering versteht man das wechselseitige Benutzen von zwei Speicherbereichen. Bei der Verwendung des EDMA eignet sich dieses Verfahren gut, um die Daten zwischen dem EDMA Controller und der CPU auszutauschen. Während der EDMA Daten in den „Ping“ Buffer schreibt, kann die CPU die Daten im „Pong“ Buffer bearbeiten. Anschließend werden die Buffer ausgetauscht. So erhält die CPU neue Daten zur Verarbeitung, während der EDMA Controller parallel neue Daten speichern kann.
Dabei muss ein „handshake“ Verfahren verwendet werden, so dass sichergestellt wird, dass die CPU die Bearbeitung der Daten beendet hat, bevor diese neue bekommt. Dazu wird in der Interrupt Routine eine Semaphore gesetzt, wenn N neue Abtastwerte zur Verfügung stehen. Im Hauptprogramm wird diese Semaphore abgefragt. Sind neue Werte vorhanden, werden die Buffer „Ping“ und „Pong“ ausgetauscht und die Semaphore zurückgesetzt, so dass der EDMA Controller neue Werte aufnehmen kann.
Um den EDMA Controller so zu konfigurieren, dass er abwechselnd in beide Buffer schreibt, muss im Register „link adress“ die Adresse des jeweils andern Buffers eingetragen werden.
Diese Technik kann durch das so genannte „Triple-Buffering“ noch weiter verbessert werden. Dabei gibt es drei Buffer für die Eingangswerte. In der Interrupt Routine werden die Puffer A und B abwechselnd mit neuen Werten beschrieben, während am Anfang des Hauptprogramms der Puffer mit den neuen Werten mit dem Puffer C ausgetauscht wird. Benötigt nun das Hauptprogramm länger als der EDMA Controller, so kann dieser trotzdem ohne Verzögerung Werte in den nächsten Buffer schreiben. Ist ein Schleifendurchlauf nun beendet, werden im Hauptprogramm wieder die beiden Buffer ausgetauscht. So sind jederzeit neue Werte für die Berechnung vorhanden und damit ist ein reibungsloser Ablauf sichergestellt, auch wenn das Hauptprogramm mehr Zeit benötigt als das „Sampeln“ neuer Werte.

4.4 C-Code Optimierung

Um die Ausführungsgeschwindigkeit des Programmcodes zu optimieren, gibt es mehrere Möglichkeiten. Nachdem das Programm getestet wurde und sichergestellt ist, dass es zuverlässig funktioniert, kann mit der Optimierung begonnen werden. Dazu wird der Code in drei Abschnitte unterteilt: zu Beginn der Prolog, anschließend der Hauptteil und der Epilog. Da das Programm die Berechnung der FFT kontinuierlich ausführt, wird der Hauptteil des Programms bei jeder Berechnung erneut aufgerufen, wobei der Prolog und der Epilog nur einmal am Anfang bzw. am Ende des Programms ausgeführt werden. Somit sind diese von geringer Bedeutung für die gesamte Programmlaufzeit.
Abbildung 4.12 zeigt die Schritte zur Reduzierung der Laufzeit. Nachdem sichergestellt ist, dass das Programm wie gewünscht funktioniert, werden zu Beginn der Optimierung die Laufzeiten der einzelnen Abschnitte mit Hilfe des „Profilers“ gemessen. Anschließend werden für die Berechnung unnötige Zwischenschritte entfernt. Dadurch wird der Programmablauf unverständlich. Deshalb ist es notwendig, das Programm vorher ausgiebig zu testen. Anschließend werden, wenn möglich, Intrinsic-Funktionen zur Berechnung eingesetzt.
Nach den einzelnen Schritten wird erneut die Programmlaufzeit gemessen, um die Auswirkung der durchgeführten Änderung bewerten zu können. Im Anschluss werden die einzelnen Compiler-Optimierungsstufen untersucht. Im letzten Schritt werden die rechenaufwändigen Programmteile in Assembler-Code umgesetzt und von Hand optimiert, um so einen möglichst effizienten Code zur Berechnung der FFT zu erhalten.

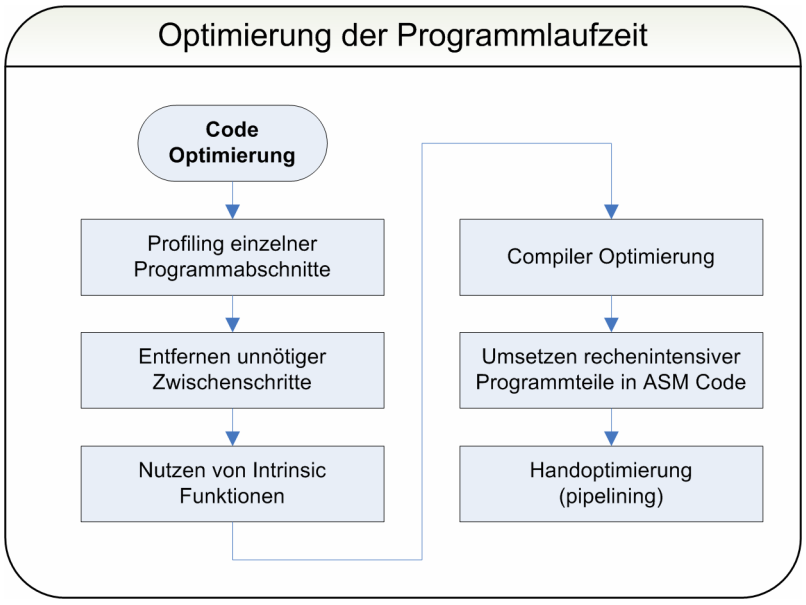


Abbildung 4.12 Optimierung der Programmlaufzeit

Die Laufzeitmessung zu Beginn hat ergeben, dass unter anderem die Berechnung des Betragsquadrates einen großen Teil der Rechenzeit in Anspruch nimmt.

```
for (i=0; i<N; i++)
      out_buf[i] = sqrt(x[2*i]*x[2*i] + x[2*i+1]*x[2*i+1]);
```

Dies ist im Wesentlichen auf die Wurzelfunktion („sqrt( )“) zurückzuführen, da diese für den Prozessor nur kompliziert umzusetzen ist. Jedoch kann diese Operation weggelassen werden, ohne das Ergebnis zu verfälschen. Somit wird das Ergebnis für das Ausgangssignal wie folgt berechnet:

```
for (i=0; i<N; i++)
      out_buf[i] = x[2*i]*x[2*i] + x[2*i+1]*x[2*i+1];
```

Dadurch verändert sich lediglich die Amplitude des Ausgangssignals.
Um die Ausführungszeit weiter zu verbessern, werden die verwendeten Schleifen ausgerollt (engl. loop unrolling), um somit einen „overhead“ und unnötiges „branchen“ zu vermeiden. Anschließend werden die Schleifen so umgeschrieben, dass diese rückwärts zählen. Dies ermöglicht dem Compiler eine effizientere Umsetzung in Assembler-Code.
Für eine bessere Verständlichkeit des Programmcodes wurde der Datentyp „Complex“ definiert, in dem die Werte nach Real- und Imaginärteil gespeichert werden können.

```
typedef struct Complex_tag {float re,im;}Complex;
```

Das Array vom Datentyp „Complex“ der Länge N wird durch ein Array der Länge $2 \cdot N$ ersetzt, indem der Realteil in den geraden Indexelementen und der Imaginärteil in den ungeraden Indexelementen gespeichert wird.
Diese Optimierungen führen zu einer Reduzierung der gesamten Programmlaufzeit von ca.12%, wobei das Speichern der komplexen Werte in ein Array der Länge $2 \cdot N$ den größten Einfluss hat.

4.4.1 Intrinsic-Funktionen

Der Compiler unterstützt Intrinsic-Funktionen[5]. Dies ermöglicht es im C-Code komplexe Rechenoperationen direkt in Assembler umzusetzen. Dazu gehören zum Beispiel Multiplikationen und Additionen.
Diese werden zur Berechnung des Butterflys eingesetzt, da dort die Koeffizienten mit den Twiddle-Faktoren multipliziert und anschließend miteinander addiert werden.
Die Multiplikationen werden durch die Intrinsic-Funktion „_mpy(src1, src2)“ ersetzt, welche vom Compiler direkt in den Assemblerbefehl „*MPY*“ umgesetzt wird.

[5] prozessorspezifische Operationen, die in Funktionsaufrufe gekapselt werden

Die anschließende Zeitmessung hat ergeben, dass sich durch die Nutzung der Intrinsic-Funktionen kein wesentlicher Geschwindigkeitsvorteil gegenüber den einfachen Multiplikationen und Additonen ergeben hat.

4.4.2 Compiler-Optimierung

Der Compiler ist in der Lage, verschiedene Optimierungen in unterschiedlichen Stufen durchzuführen. Codespezifische Optimierungen („high-level“) werden im „Optimizer“ und hardwarespezifische Optimierungen („low-level“) werden im Code-Generator umgesetzt. Abbildung 4.13 [20] zeigt ein Ablaufdiagramm des Compilers mit Optimierung und Codeerzeugung.

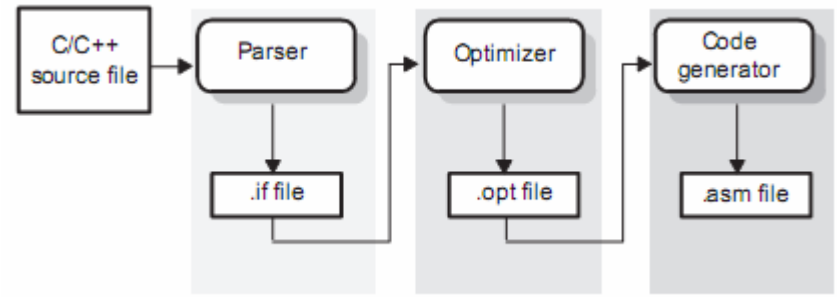


Abbildung 4.13 Compiliervorgang mit Optimierung

Der Compiler bietet mehrere Optimierungsstufen (Optimierungslevel), in denen der Code hinsichtlich einer geringen Laufzeit optimiert wird. Die unterschiedlichen Optimierungsstufen (o0–o3) mit den durchgeführten Modifikationen sind im Folgenden aufgeführt:

- Optimierungslevel „o0“
  - Vereinfachung des Kontrollflusses
  - Variablen feste Register zuweisen
  - Rückwärtslauf der Schleifen
  - Eliminieren von überflüssigem Code
  - Vereinfachung von Anweisungen und Ausdrücken

- Optimierungslevel „o1“ wie unter „o0“ plus:
  - Entfernen von überflüssigen Anweisungen
  - Löschen von gemeinsamen lokalen Ausdrücken

- Optimierungslevel „o2“ wie unter „o1“ plus:
  - Software-Pipelining
  - Schleifen-Optimierung

- Eliminierung identischer Ausdrücke (common subexpression elimination)
- Löschen von globalen ungenutzten Variablen
- Konvertierung von Arrayreferenzen in Schleifen
- Schleifen ausrollen (loop unrolling)

- Optimierungslevel „o3“ wie unter „o2“ plus:
  - Löschen von nicht aufgerufenen Funktionen
  - Rückführung von kurzen Funktionsaufrufen in das Hauptprogramm
  - Umsortierung der Funktionsdeklarationen
  - Speichern des Funktionsarguments in den Funktionsaufruf, wenn möglich

Die oben beschriebenen Optimierungen werden ausschließlich vom „Optimizer“ durchgeführt. Der Code-Generator führt davon unabhängig weiter hardwarespezifische Optimierungen durch. Dieser Optimierungsschritt wird immer ausgeführt, ist jedoch effizienter, wenn zuvor der „Optimizer“ benutzt wird [20].
Wie zuvor beschrieben, führt der „Optimizer“ eine Reihe von Optimierungen durch. Wie oben zu sehen ist, gehören dazu auch das Ausrollen und Rückwärtszählen von Schleifen sowie das Speichern viel genutzter Werte in Register. Dadurch ergibt sich für die in Kapitel 4.4 beschriebene Handoptimierung kein Geschwindigkeitsvorteil, wenn die Compiler-Optimierung aktiviert ist, da der „Optimizer“ auch die per Hand vorgenommenen Änderungen durchführt, um die Programmlaufzeit zu verkürzen.
Des Weiteren führt der „Optimizer“ ab dem Optimierungsschritt „o2“ ein „Software-Pipelining“ des Programmcodes durch. Da der Prozessor zwei Datenpfade mit jeweils vier arithmetischen und logischen Einheiten besitzt (s. Abbildung 4.17), wird der Code so optimiert, dass möglichst viele Rechenoperationen parallel durchgeführt werden können (theoretisch acht Operationen pro Takt). Diese Optimierung führt zu einer erheblichen Reduzierung der Rechenzeit, wie sich schon in Abbildung 4.10 durch die Nutzung der „o3“ Optimierung gezeigt hat.
Das Parallelisieren der Rechenoperationen wird vom „Optimizer“ durchgeführt. Jedoch lässt sich der Code oft noch weiter parallelisieren, wenn dieser direkt in Assembler geschrieben und von Hand optimiert wird. Die Vorgehensweise für diese Optimierung wird im Kapitel 4.5 beschrieben.

4.4.3 2N Punkte Transformation

Eine Möglichkeit, die Ausführungszeit weiter zu verkürzen, ist, die gegebene *N* Punkte FFT zu nutzen um damit, wie in Kapitel 3.8 beschrieben, eine *2N* Punkte FFT zu berechnen.

Dazu wird zunächst die doppelte Anzahl an Abtastwerten aufgenommen. Diese werden dann wie folgt auf den Eingangsvektor der FFT geschrieben:

```
for (i=0; i<N; i++){
      x[2*i]   =in_buf[2*i];
      x[2*i+1] =in_buf[2*i+1];
}
```

Somit wird jeder zweite Abtastwert auf den Realteil und jeder nächste auf den Imaginärteil des Eingangsvektors geschrieben. Bei der herkömmlichen Implementierung wird der Imaginärteil des Eingangsvektors zu null gesetzt und somit nicht genutzt.
Mit diesem Eingangsvektor wird nun wie bisher die FFT berechnet. Am Ende der Berechnung muss aus dem Ausgangsvektor, wie zuvor beschrieben, das Ergebnis bestimmt werden. Mit Hilfe dieses Verfahrens können z.B. 1024 Punkte mit einer 512 Punkte FFT berechnet werden.
Abbildung 4.14 zeigt den Programmablauf mit allen nötigen Rechenschritten zur Berechnung einer *N* Punkte FFT mit *2N* Werten. Wie schon zuvor beschrieben, werden zunächst die aufgenommenen Eingangswerte auf den x-Vektor zur FFT Berechnung aufgeteilt. Anschließend wird wie in den Programmen zuvor die Berechnung durchgeführt. Daraufhin wird aus dem Ergebnisvektor $X(k)$ der Vektor $X^*(<-k>_N)$ berechnet ($X^*(<-k>_N)$ ist in Gleichung (3.30) definiert). Aus diesen beiden Vektoren werden im nächsten Schritt $G(k)$ und $H(k)$ bestimmt. Daraus lässt sich wiederum mit Hilfe von W_N^k der Vektor $V(k)$ berechnen. Zum Schluss steht nach der Bildung des Betragsspektrums das Ausgangsspektrum der *2N* Punkte FFT zur Verfügung.

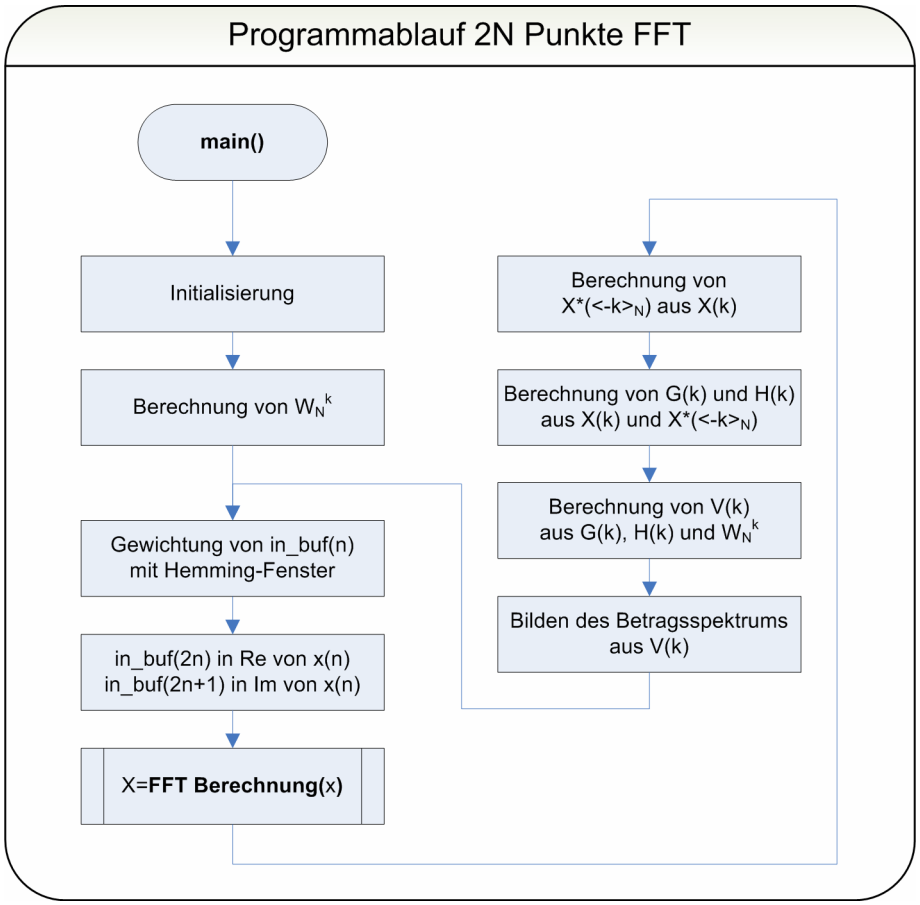


Abbildung 4.14 Programmablauf 2N Punkte FFT

Diese Art der Umsetzung verspricht eine große Zeitersparnis, da sich *2N* Punkte mit einer *N* Punkte FFT berechnen lassen. Allerdings relativiert sich die Ersparnis durch die Nachbereitung der Werte. Ein Vergleich der *2N* Punkte FFT mit der Standart FFT aus Abschnitt 4.2.1 ist in Abbildung 4.15 dargestellt. Es ist zu sehen, dass eine Nutzung dieses Verfahrens für große *N* eine Ersparnis von einigen Millisekunden einbringt.

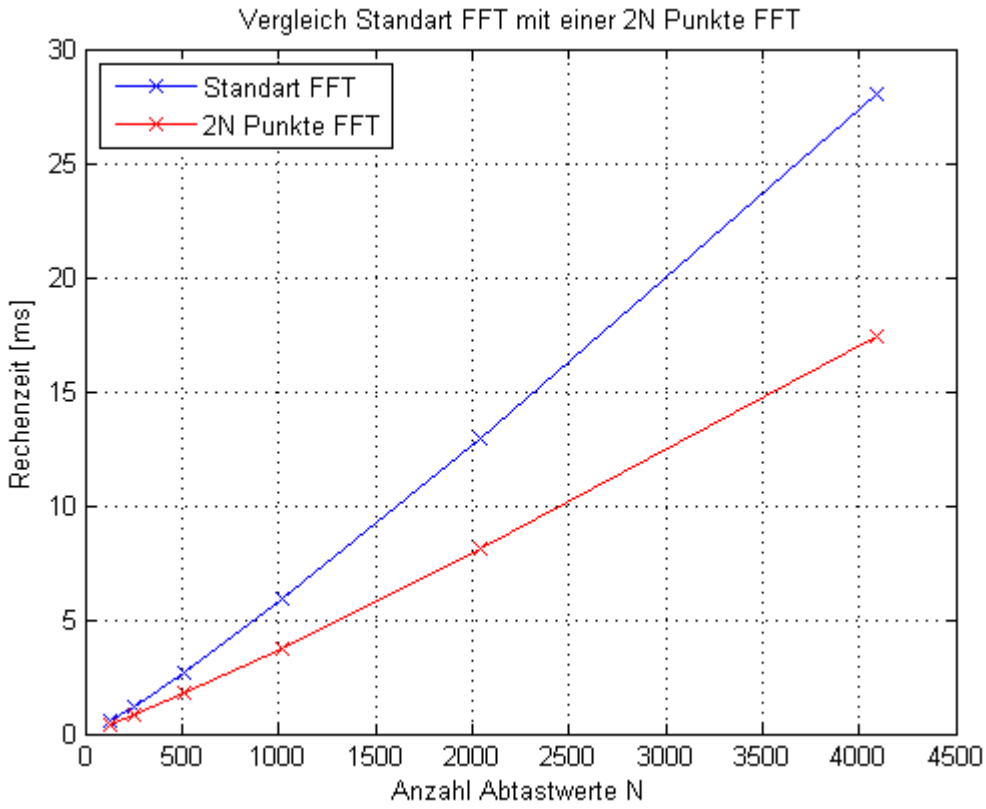


Abbildung 4.15 Vergleich einer Standard FFT mit einer 2N Punkte FFT

Um die Effizienz des Programms zu steigern, wird der C-Code so optimiert, dass die gesamte Nachbereitung der Werte in einem Schleifendurchlauf stattfindet. Dafür ist es wichtig, das Programm zuvor ausreichend zu testen, da der Programmablauf anschließend nur schwer nachzuvollziehen ist.

Wie in Tabelle 4.3 zu sehen, verkürzt sich die Rechenzeit durch die Optimierung des C-Codes. Jedoch bringt erst die Compiler-Optimierung (o3) einen wesentlichen Geschwindigkeitsvorteil gegenüber dem nicht optimierten Code. Abbildung 4.16 zeigt die in der Tabelle dargestellten Ergebnisse.

Tabelle 4.3 Laufzeiten der 2N Punkte FFT

	nicht optimiert	C-Code optimiert	o3 optimiert	o3 und C-Code optimiert
2*N	Zeit [ms]	Zeit [ms]	Zeit [ms]	Zeit [ms]
128	0.38	0.31	0.16	0.13
256	0.82	0.69	0.34	0.29
512	1.76	1.51	0.73	0.36
1024	3.76	3.26	1.56	1.36
2048	8.12	7.1	3.36	3.0
4096	17.4	15.6	7.4	6.6

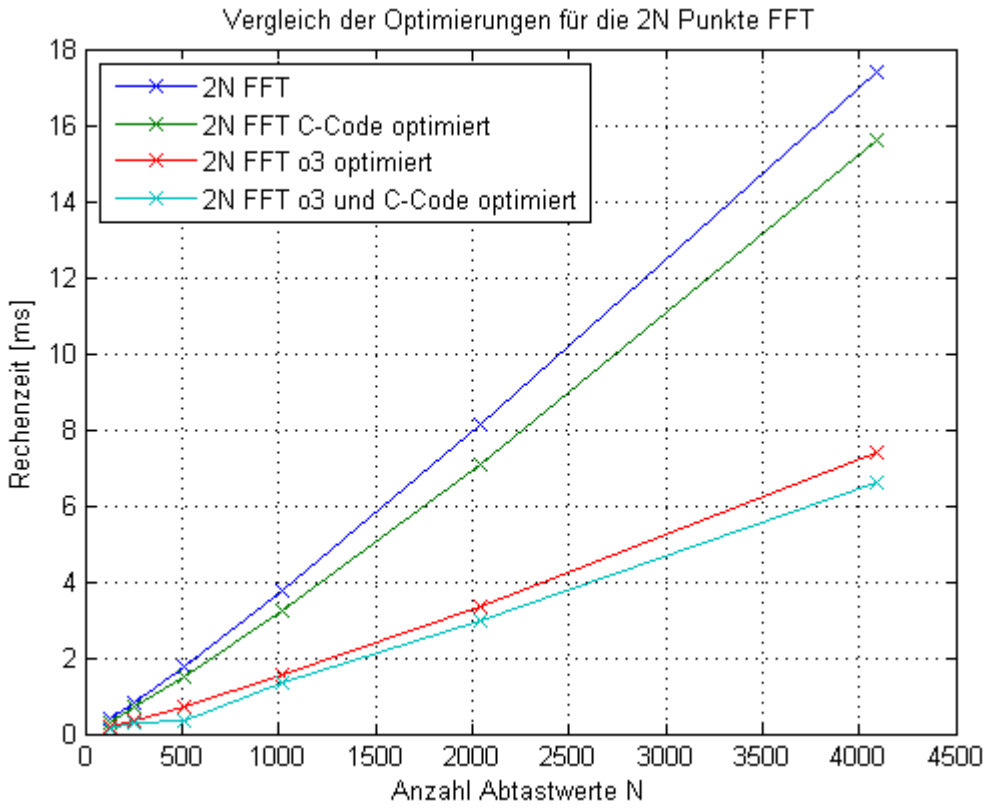


Abbildung 4.16 Vergleich der Optimierungen für die 2N Punkte FFT

4.5 Assembler Code Optimierung

Wie sich im Laufe der Arbeit gezeigt hat, lässt sich die Ausführungszeit für die FFT Berechnung durch verschiedenste Optimierungen um einige Millisekunden verkürzen. Eine große Geschwindigkeitssteigerung zeigt sich bei der Parallelisierung der Rechenschritte durch den Compiler (Pipelining). Um den Code weiter zu verbessern, ist es sinnvoll, Programmteile direkt in Assembler zu schreiben und den Code anschließend von Hand zu optimieren, so dass möglichst viele Operationen parallel ausgeführt werden können.

Um den C-Code in Assembler-Code umzusetzen, ist eine genaue Kenntnis der Hardwarestruktur sowie die Kenntnis der Assembler-Befehle für diese Hardware notwendig.

Die Programmierung in Assembler findet auf Registerebene statt. Im Gegensatz zur Programmierung in C können für den Programmablauf wichtige Registerinhalte, wie zum Beispiel der „Programm Counter“, überschrieben werden und so kann durch eine fehlerhafte Programmierung das Programm zum Absturz gebracht werden.

Dieser Zugriff auf die Registerebene hat den Vorteil, dass die Anweisungen direkt in einen ausführbaren Maschinencode übertragen werden. Jedoch ist ein Assembler-Code im Gegensatz zum C-Code speziell auf eine bestimmte Hardware ausgelegt und somit nicht kompatibel. Dies ermöglicht eine besonders effiziente Programmierung durch das Parallelisieren der

Rechenoperationen. Eine Übersicht über den Befehlssatz für den „TMS320C6713“ befindet sich in Anhang B [24].
Der „TMS320C6713“ verfügt über 32 Register, die in zwei Datenpfade (A und B) unterteilt sind (s. Abbildung 4.17 [21]). Jeder dieser Pfade besitzt eine .S, .L, .M und .D Einheit, welche die in Anhang B dargestellten Operationen durchführen können. Der Prozessor ist zu 100% ausgelastet, wenn in jedem Takt alle acht Recheneinheiten benutzt werden.
Dies ist in der Praxis nur schwer zu erreichen, jedoch ist ein handoptimierter Code häufig schneller als ein vom Compiler optimierter Code.

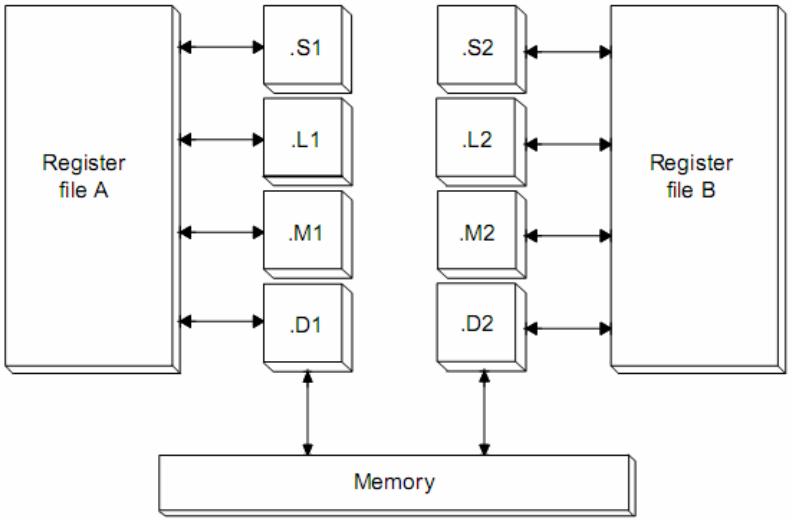


Abbildung 4.17 Funktionseinheiten des TMS320C6713

4.5.1 Implementierung der Radix 2 FFT

Von der Firma *Texas Instruments* wurde 1998 ein handoptimierter Assemblercode zur Lösung der Radix 2 FFT DIT für Floating-Point Datentypen entwickelt, dieser Code ist speziell auf den „TMS320C6713“ Prozessor ausgelegt. Das Programm wird im Folgenden genutzt, um die FFT zu berechnen.
Der Code verspricht eine besonders effiziente Berechnung, da der Prozessor abgesehen vom Prolog und Epilog bei der Berechnung durch „Software-Pipelining“ zu 90% ausgelastet ist. Eine solche Auslastung lässt sich nur durch eine Handoptimierung erreichen.
Der Programmablauf der Berechnung ist identisch mit dem Vorherigen. Lediglich der Funktionsaufruf „radix2( )“ (vgl. Abbildung 4.5) wird durch den Aufruf der Assembler-Funktion ersetzt.
Durch die Nutzung dieses Assemblercodes verkürzt sich die Berechnungszeit für die FFT. *Texas Instruments* gibt die Rechenzeit der Radix 2 Funktion mit $\left[(2 \cdot N) + 23\right] \cdot \log_2(N) + 6$ Taktzyklen an. Daraus ergibt sich für die Anzahl der betrachteten Eingangswerte N und ei-

nem CPU Takt von 225 MHz die in Tabelle 4.4 dargestellte, theoretische Ausführungszeit. In der rechten Spalte der Tabelle ist die tatsächliche Laufzeit des Programms dargestellt.

Tabelle 4.4 Laufzeit der Assembler optimierten Radix 2 FFT

N	Cycles	Theo. Zeit [µs]	Zeit [µs]
64	912	4.05	4.4
128	1959	8.71	9
256	4286	19.05	21
512	9429	41.91	54.8
1024	20716	92.07	154
2048	45315	201.40	333
4096	98586	438.16	706

Bei den Messergebnissen ist zu beachten, dass für einen gesamten Programmdurchlauf mehr Rechenzeit notwendig ist, da die Eingangswerte zunächst vom Datentyp „integer“ auf den Datentyp „float“ umgewandelt werden und nach der FFT das Spektrum berechnet werden muss. In der Tabelle ist nur die benötigte Rechenzeit für die Assembler Routine dargestellt. Die Vor- und Nachbereitung nimmt im Vergleich zur eigentlichen Berechnung viel Zeit in Anspruch.
Bei den Messergebnissen ist auffällig, dass für große *N* die gemessenen Werte immer weiter von den theoretisch berechneten Zeiten abweichen. Ein Grund dafür ist, dass nicht alle zur Berechnung nötigen Werte im Level 1 Cache gespeichert werden können, da dieser nur 4 kByte umfasst. Für eine „in place“ Berechnung einer Floating-Point FFT ist mindestens der in Tabelle 4.5 dargestellte Speicherplatz notwendig.

Tabelle 4.5 Speicherplatzbedarf einer Radix 2 FFT

N	Radix 2 FFT	
64	768	Byte
128	1536	Byte
256	3072	Byte
512	6144	Byte
1024	12288	Byte
2048	24576	Byte
4096	49152	Byte

Aus der Tabelle ist ersichtlich, dass im Level 1 Cache maximal für eine 256 Punkte FFT genügend Speicherplatz vorhanden ist. Für eine FFT mit mehr als 256 Punkten muss somit zwangsläufig auch der Level 2 Cache verwendet werden. Für den Zugriff auf diesen Speicher benötigt der Prozessor mehr Zeit als für den Zugriff auf den Level 1 Cache (vgl. Abbildung 2.3). Somit ergibt sich eine längere Programmlaufzeit im vergleich zu den von *TI* angegebenen Werten.

Des Weiteren ist es durch die Nutzung der handoptimierten FFT Funktion nicht mehr sinnvoll, mit dem derzeitigen C Code *2N* Punkte mit einer *N* Punkte FFT zu berechnen. Da die Umrechnung der Werte, die nach der Berechnung notwenig ist, mehr Zeit in Anspruch nimmt als die direkte Berechnung einer FFT mit der doppelten Anzahl von Eingangswerten.

4.5.2 Implementierung der Radix 4 FFT

Neben der handoptimierten Radix 2 FFT DIT Routine gibt es von *Texas Instruments* eine ebenfalls handoptimierte Radix 4 FFT DIF für den „TMS320C6713“ Prozessor in Floating-Point Arithmetik. *TI* gibt für die Rechenzeit dieser Routine $\left[(14 \cdot N/4) + 23\right] \cdot \log_4(N) + 20$ Takte an. Dadurch ergibt sich für 1024 Werte eine theoretische Verkürzung der Rechenzeit um 11,8 µs im Vergleich zur Radix 2 FFT. Allerdings hat die Verwendung dieser Funktion den Nachteil, dass die Anzahl der Eingangswerte einer Potenz der Basis 4 entsprechen muss. Somit ist die Funktion nicht so flexibel einsetzbar wie die Radix 2 FFT.
Für die theoretischen Rechenzeiten ergeben sich die in Tabelle 4.6 dargestellten Werte.

Tabelle 4.6 Laufzeit der Assembler optimierten Radix 4 FFT

N	Cycles	Theo. Zeit [µs]	Zeit [µs]
64	761	3.38	3.65
256	3696	16.43	17.6
1024	18055	80.24	188
4096	86174	383.00	983

Wie in der Tabelle zu sehen ist, weichen die gemessenen Laufzeiten für große *N* erheblich von den theoretischen Werten ab. Der Grund dafür ist wie schon bei der Radix 2 FFT, dass nicht genügend Speicherplatz im Level 1 Cache vorhanden ist. Bei diesem Programm wirkt sich das besonders stark aus, weil aufgrund der größeren Komplexität der Radix 4 FFT nicht genügend Register vorhanden sind und deshalb Zwischenergebnisse während der Berechnung gespeichert werden müssen. Dadurch erhöht sich die Zahl der Speicherzugriffe und daraus resultierend die Programmlaufzeit.

4.5.3 Betrachtung der Radix 8 FFT

Wie in den beiden vorhergehenden Abschnitten beschrieben, lässt sich der theoretische Geschwindigkeitsvorteil von schnellen Fourier-Transformationen mit hohen Basen nicht ausnutzen, da die Programmstruktur aufwändiger wird und dadurch für eine direkte Berechnung nicht genügend Register zur Verfügung stehen. Dies erfordert ein Zwischenspeichern der Ergebnisse und damit mehr Speicherzugriffe, welche die Ausführung verlangsamen, insbesondere, wenn die Speicherzugriffe auf den Level 2 Cache getätigt werden müssen. Davon abge-

sehen ist für eine Implementierung einer solch komplexen Assemblerroutine viel Erfahrung in der Assemblerprogrammierung und Zeit notwendig.

4.5.4 2N Punkte Transformation

Die Nutzung der *2N* Punke Transformation ist nur sinnvoll, wenn sich dadurch die Rechenzeit verkürzt. Durch die Nutzung des optimierten Assemblercodes von *Texas Instruments* ist es schneller, eine FFT mit z.B. 1024 Punkten zu berechnen, als durch Nutzung der *2N* Punkte Transformation eine 512 Punkte FFT und somit 1024 Werte zu berechnen. Der Grund dafür ist die Rechenzeit, die der C-Code benötigt, um das Ergebnis der *2N* Punkte FFT zu berechen. Um diese Rechenzeit zu verkürzen, wird dieser Teil des C-Programms in Assembler implementiert und versucht zu optimieren.
Die Funktion „calc.asm“ führt die Berechnung der *2N* Ausgangswerte aus dem Ergebnis der FFT durch. Abbildung 4.18 zeigt das Flussdiagramm des Programms. Dabei stellt die linke Seite in der Abbildung den Datenpfad A und die rechte Seite den Datenpfad B dar. Jeder Kasten im Diagramm entspricht einem Takt während des Programmablaufs, in dem die jeweilige Operation ausgeführt wird. Die Takte, in denen der Prozessor keine Operation ausführt, sind nicht dargestellt.

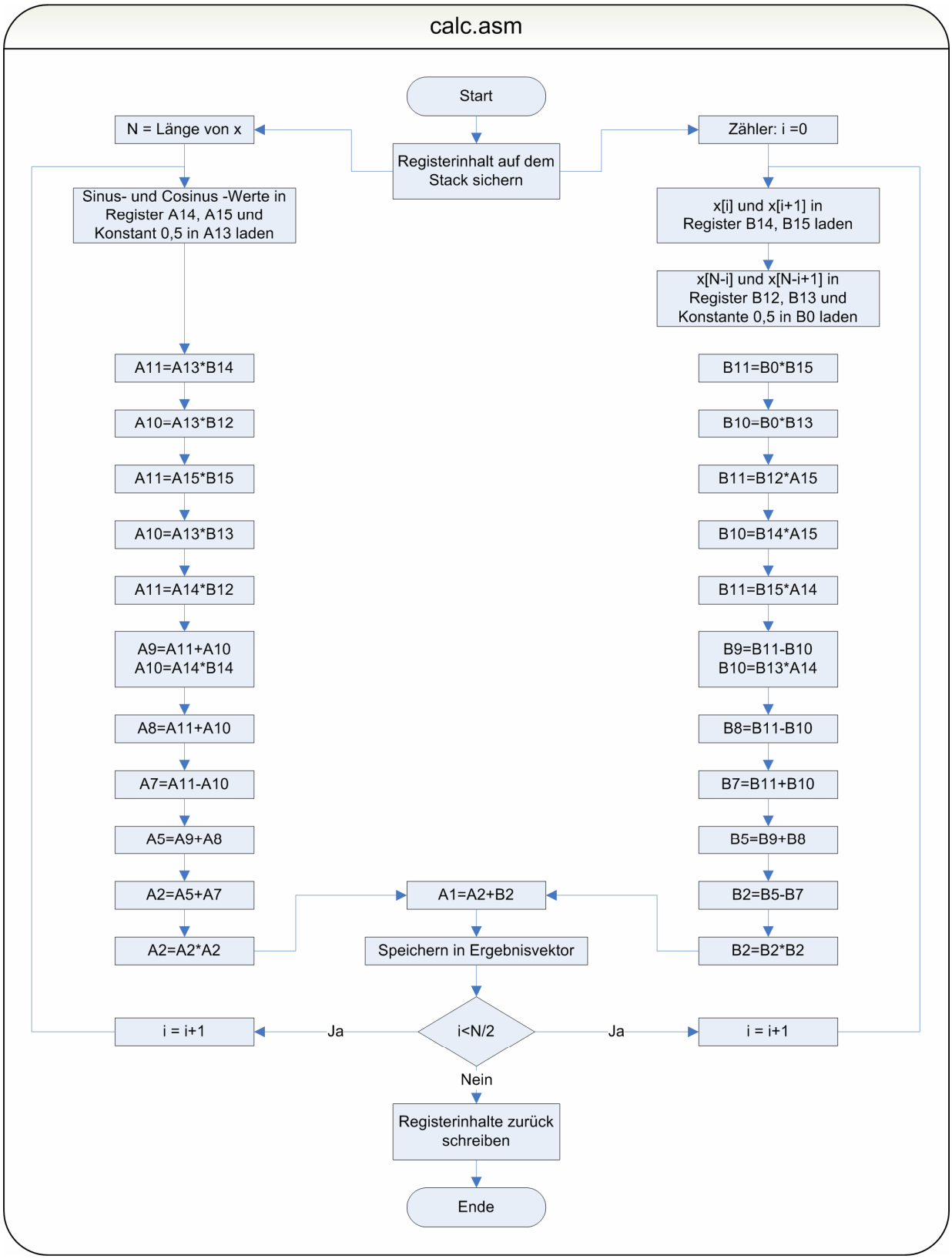


Abbildung 4.18 Programmablauf calc.asm

Vor Beginn der eigentlichen Berechnung wird der Inhalt wichtiger Register, wie z.B. des „Programm Counters“, auf dem „Stack“ gesichert, damit für die Berechnung alle Register zur Verfügung zu stehen. Dabei muss darauf geachtet werden, dass während der Programmlauf-

zeit kein Interrupt auftreten kann, da dies aufgrund des fehlenden „Programm Counters“ zum Absturz des Programms führen würde.
Anschließend wird die Anzahl der Abtastwerte in ein Register geschrieben und somit der Zähler initialisiert, welcher die Abbruchbedingung des Programms festlegt. Daraufhin werden die vor Beginn des Programms berechneten Twiddle-Faktoren (Sinus- und Cosinus - Werte) (s. Abbildung 4.14) und die Ergebnisse der FFT Berechnung in die Register geladen. Dann beginnt die Berechnung der Ergebnisse nach den Formeln aus Kapitel 3.8. Zum Schluss wird das Betragsspektrum gebildet und das Endergebnis gespeichert.
Laufzeitmessungen haben ergeben, dass der Assembler-Code zwar einen Geschwindigkeitsvorteil von ca. 500 µs (für 2048 Eingangswerte) gegenüber dem optimiertem C-Code bietet, er aber noch immer langsamer ist als eine direkte Berechnung der FFT mit $2 \cdot N$ Werten. Daher ist es noch immer nicht lohnenswert, die *2N* Punkte-Transformation einzusetzen.
Um die Programmlaufzeit weiter zu verkürzen, wird der Assembler-Code so umgeschrieben, dass möglichst viele Instruktionen parallel abgearbeitet werden (Software-Pipelining). Dazu muss der Ablauf genau geplant werden, so dass die Daten zum richtigen Zeitpunkt weiterverarbeitet werden können. Dies geschieht mit Hilfe eines Ablaufplans (s. Anhang D). In einem solchen Plan wird jeder Takt eines Schleifendurchlaufs mit den ausgeführten Rechenoperationen grafisch dargestellt. Dieser Programmdurchlauf wird daraufhin dupliziert, so dass zwei Programmdurchläufe hintereinander vorliegen. Im nächsten Schritt werden beide ineinander geschoben, so dass sich die beiden Darstellungen überschneiden. Dieser Vorgang wird mit dem nächsten Durchlauf so lange wiederholt, bis möglichst viele der acht Recheneinheiten genutzt werden. Dadurch entstehen drei Programmteile: Prolog, Hauptteil (Kernel) und Epilog. Zu Beginn sind einige Takte notwendig, bis alle Daten in den Registern zur parallelen Bearbeitung vorliegen. Ist dies der Fall, dann werden im Hauptteil möglichst viele Instruktionen gleichzeitig ausgeführt. Der Hauptteil wird entsprechend der Anzahl an Eingangswerten so lange wiederholt, bis alle Berechnungen durchgeführt sind. Im Epilog findet anschließend das „Aufräumen“ statt. Es werden alle Instruktionen bearbeitet, die sich nicht im Hauptteil durchführen lassen, wie z.B. das Speichern der zuletzt berechneten Ergebnisse.
Bei der Erstellung des Ablaufplans hat sich gezeigt, dass nicht das gesamte Programm (s. Abbildung 4.18) sinnvoll parallelisiert werden kann, da dadurch der Hauptteil sehr lang werden würde und dafür nicht genug Register zur Verfügung stehen. Deshalb wird das Programm in zwei Programme unterteilt. Zum einen in das Programm „calc_piped.asm“, in dem ausschließlich die Berechnung der *2N* Werte für die FFT vorgenommen wird, und zum anderen in das Programm „sum.asm“, in dem das Betragsspektrum der FFT gebildet wird.
Der Funktion „calc( )“ wird ein Pointer auf das erste Element der Sinus- und Cosinuswerte, sowie die Adresse des ersten und letzen Arrayelements der FFT Berechnung, übergeben. Außerdem wird die Adresse eines Arrays übergeben, in dem die Zwischenergebnisse gespeichert werden sollen.

```
calc(&sincos[0],&x[0],&x[2*N],&temp[0]);
```

Das Array mit den Zwischenergebnissen wird nach der Berechnung zusammen mit der Adresse des ersten und letzen Elementes des Ergebnisvektors und der halben Anzahl an Eingangswerten an die Funktion „sum( )“ übergeben.

```
sum(&temp[0], &out_buf[1],&out_buf[2*N], N);
```

Mit Hilfe dieser Funktionen ergeben sich die in

Tabelle 4.7 dargestellten Werte für die Berechnung der *2N* Punkte.

Tabelle 4.7 Laufzeit des 2N Punkte Algorithmus in Assembler

2*N	Cycles	Theo. Zeit[µs]	Zeit [µs]
128	1035	4.60	4.76
256	2203	9.79	9.92
512	4504	20.02	21.2
1024	9605	42.69	44.4
2048	20129	89.46	90.8
4096	40450	179.78	182

4.5.5 Typkonvertierung

Laufzeitmessungen des Programmcodes haben ergeben, dass die Typkonvertierung (cast) vom Datentyp „integer“ auf den Datentyp „float“ und das damit verbundene Umspeichern einen großen Teil der gesamten Programmlaufzeit benötigt. Daher ist es sinnvoll, den Programmcode zu optimieren.

```
for (i=0; i<N; i++){
      x[2*i]   =(float)in_buf[2*i];
      x[2*i+1]=(float)in_buf[2*i+1];
}
```

Dazu wird das Programm „int_to_float.asm“ erstellt. Dies kann aus dem Hauptprogramm mit den entsprechenden Übergabeparametern aufgerufen werden.

```
int_to_float(float x, int in_buf, N);
```

Der erste Parameter ist ein Pointer auf das Array mit den *N* „integer“ Werten. Der zweite Parameter ist ein Pointer auf ein „float“ Array, in dem die Daten gespeichert werden, und der dritte Parameter ist die halbe Anzahl der umzuwandelnden Werte.

Ein Ablaufplan des Programms ist in Abbildung 4.19 dargestellt. Zu Beginn des Programms werden zwei Werte in die Register geladen. Da es sich bei den abgetasteten Werten um 32 Bit „integer“ Werte handelt, in denen die oberen 16 Bit dem linken Kanal und die unteren 16 Bit dem rechten Kanal entsprechen, wird zunächst wahlweise einer der Kanäle maskiert. Anschließend werden die beiden 16 Bit Werte von x[i] und x[i+1] in 32 Bit Werte vom Datentyp „float“ umgewandelt. Danach werden diese gespeichert und der Vorgag wird so lange fortgesetzt, bis alle Elemente umgewandelt sind.

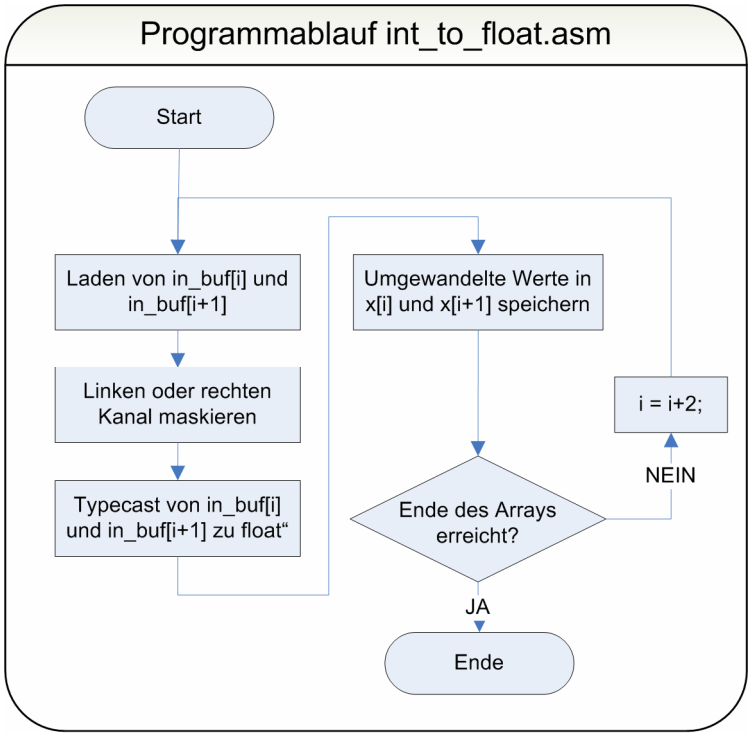


Abbildung 4.19 Programmablauf int_to_float.asm

Nach dem Funktionstest wird die Anzahl der zur Berechnung nötigen Takte mit Hilfe von Software-Pipelining weiter reduziert. Dadurch ergibt sich eine Programmlaufzeit von 23 µs für die Konvertierung von 2048 „integer“ Werten im Gegensatz zu 200 µs für einen äquivalenten optimierten C-Code.

5 Ergebnis

In dieser Arbeit wurden verschiedene Verfahren zur Berechnung einer diskreten Fourier-Transformation untersucht. Dazu erfolgte zunächst eine mathematische Betrachtung möglicher Lösungsverfahren, die eine geringe Anzahl an Rechenoperationen ermöglichen. Algorithmen, die nur in Spezialfällen eingesetzt werden können, wie z.B. der Goertzel-Algorithmus, mit dem sich nur einzelne Spektralkomponenten berechnen lassen, wurden nicht genauer untersucht, da diese keinen vielfältigen Einsatz ermöglichen. Nach den theoretischen Untersuchungen wurden die Unterschiede zwischen einer Implementierung der FFT in Gleitkommaarithmetik und Festkommaarithmetik anhand einer Radix 2 FFT DIT herausgearbeitet.
Im Anschluss daran wurden sowohl die Algorithmen als auch die Vor- bzw. Nachbereitung der Daten bezüglich der Rechenzeit verbessert.
Die Ergebnisse dieser Optimierung werden im Folgenden näher thematisiert.

5.1 Gegenüberstellung und Bewertung

Die Umsetzung hat gezeigt, dass sich eine FFT in Floating-Point Arithmetik einfach realisieren lässt, da es faktisch zu keinen Überläufen des Wertebereichs kommen kann. Allerdings benötigt die Fixpoint Implementierung des Programms weniger Rechenzeit, wie in Abbildung 4.9 zu sehen ist. Durch die Compiler-Optimierung reduzieren sich die Laufzeiten der Programme um mehr als die Hälfte. Durch das bei der Optimierung durchgeführte Software-Pipelining beschränkt sich der Geschwindigkeitsvorteil der optimierten Fixpoint FFT auf die für die Typumwandlung benötigte Rechenzeit.
Um diese bei der Verwendung von Floating-Point Datentypen notwendige Operation möglichst schnell durchzuführen, wird für diesen Programmabschnitt ein Assemblerprogramm geschrieben. Dieses übernimmt das Maskieren des linken bzw. rechten Kanals und die Konvertierung vom Datentyp „short“ auf den Datentyp „float“. Dadurch reduziert sich die benötigte Zeit für diesen Programmabschnitt um den Faktor acht gegenüber dem äquivalenten C-Code. Es entsteht dadurch kein Verlust an Genauigkeit, da bei der Konvertierung die gleiche Funktion genutzt wird. Die kürzere Rechenzeit ergibt sich durch ein effizientes Software-Pipelining.

Die Laufzeitmessungen haben ergeben, dass sich durch die Berechnung von *2N* Punkten mittels einer *N* Punkte FFT eine Verkürzung der Programmlaufzeit um etwa 12% ergibt (vgl. Tabelle 4.3). Um die Laufzeit des Algorithmus und damit den gesamten Programmablauf weiter zu verkürzen, wird auch dieser Teil in Assembler-Code umgesetzt. Damit viele In-

struktionen parallel ausgeführt werden können, wird die Funktion zunächst dahingehend mit Hilfe eines Programmablaufplans (s. Anhang D) optimiert und anschließend in Assembler Code umgesetzt. Damit ergeben sich die in Tabelle 5.1 dargestellten Laufzeiten. Zum Vergleich sind in der linken Spalte die Laufzeiten der C-Routine dargestellt, welche die gleiche Berechnung durchführt.

Tabelle 5.1 Vergleich der 2N Punkte Routine in C und Assembler

	C-Code	**ASM-Code**
2*N	Zeit [µs]	Zeit [µs]
128	23.6	4.76
256	47.6	9.92
512	96.3	21.2
1024	195	44.4
2048	395	90.8
4096	792	182

Abbildung 5.1 zeigt die gemessenen Laufzeiten. Blau dargestellt ist die benötigte Zeit zur Berechnung der *2N* Punkte, welche nach der *N* Punkte FFT durchgeführt wird. Für die Berechnung werden die Formeln aus Kapitel 3.8 verwendet. In Rot dargestellt ist die benötigte Zeit des Assembler Programms. Beide Programme führen die identischen Rechenoperationen durch. Die Verkürzung der Programmlaufzeit um den Faktor 4,5 wird durch eine effiziente Nutzung der zur Verfügung stehenden Recheneinheiten erreicht.

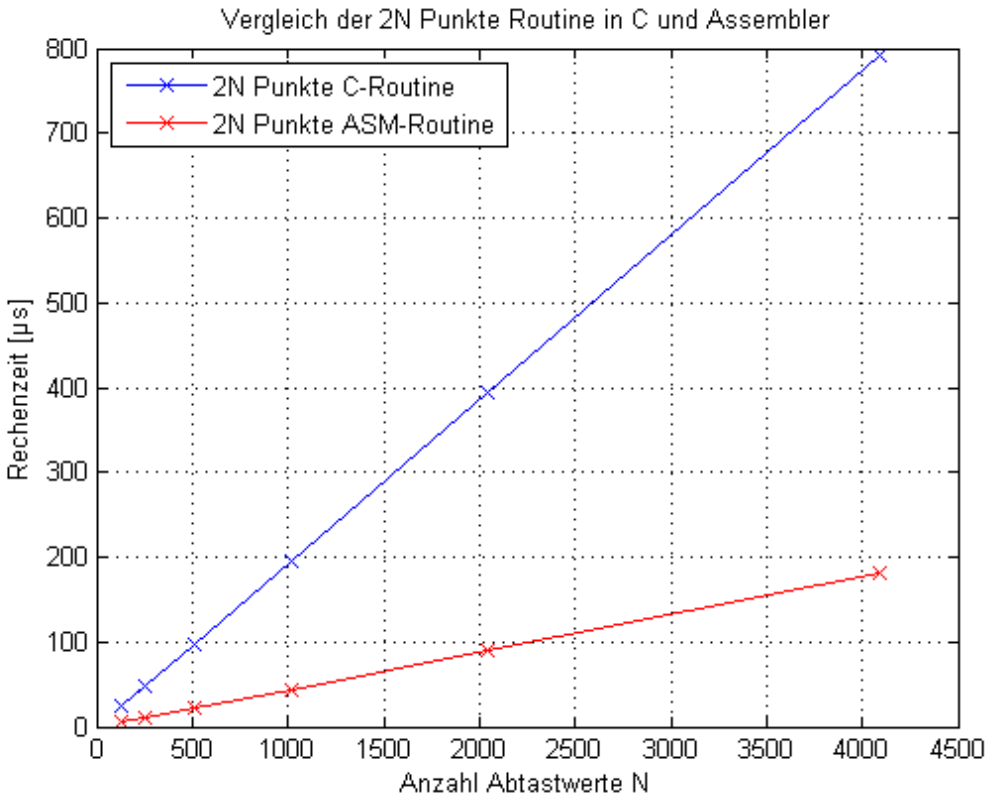


Abbildung 5.1 Vergleich der 2N Punkte Routine in C und Assembler

Durch die Verwendung des *TI* Assemblercodes zur Berechnung der Radix 2 FFT DIT und die Funktion zum Konvertieren der Eingangswerte, ergeben sich die in Tabelle 5.2 dargestellten Zeiten. Im Vergleich zum optimierten C-Code ergibt sich durch die Nutzung des Assemblercodes eine fünf Mal kürzere Laufzeit. Dieser Zusammenhang ist in Abbildung 5.2 grafisch dargestellt.

Tabelle 5.2 Vergleich zwischen C-Code und ASM-Code zur FFT Berechnung

	C-Code	**ASM-Code**
N	Zeit [µs]	Zeit [µs]
64	85	21.88
128	192	43.2
256	424	86.2
512	950	182
1024	2160	424
2048	4860	888
4096	10050	1830

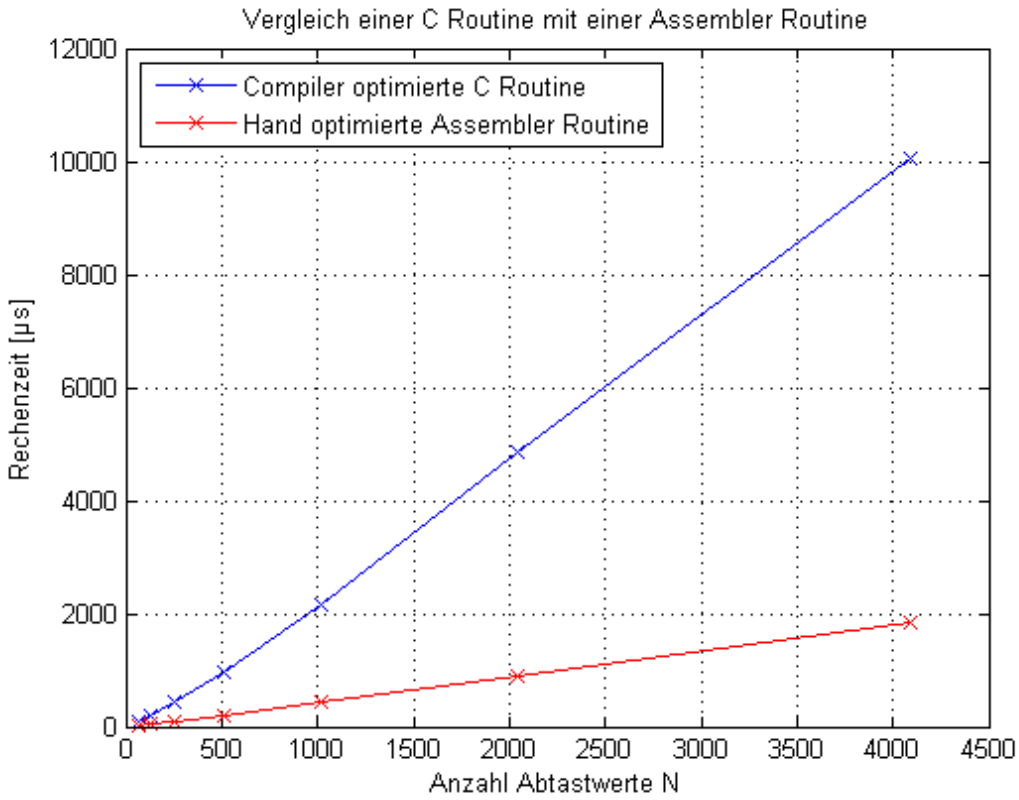


Abbildung 5.2 Vergleich der Radix 2 C Routine mit der Radix 2 ASM Routine

Trotz der Geschwindigkeitssteigerung, die sich durch die Nutzung der Radix 2 wie auch der Radix 4 FFT ergibt, werden die theoretisch angegebenen Laufzeiten der Programme für mehr als 256 Eingangswerte nicht erreicht (vgl. Kapitel 4.5.1und 4.5.2). Der Grund dafür ist, dass nicht alle Daten zur Berechnung im Level 1 Cache gespeichert werden können und dadurch auch der Level 2 Cache verwendet werden muss. Tabelle 5.3 zeigt die mittels Profiler ermittelten Laufzeiten.

Tabelle 5.3 Laufzeiten Radix 2 und Radix 4 FFT

	Radix 2 ASM		**Raidx 4 ASM**	
N	Theo. Zeit [µs]	Zeit [µs]	Theo. Zeit [µs]	Zeit [µs]
64	4.05	4.4	3.38	3.65
128	8.71	9.0		
256	19.05	21	16.43	17.6
512	41.91	54.8		
1024	92.07	154	80.24	188
2048	201.40	333		
4096	438.16	706	383.00	983

Wie in der Tabelle 5.3 und in Abbildung 5.3 zu sehen ist, benötigt die Radix 4 Routine für N größer 256 Punkte ca. 20% mehr Rechenzeit als die Radix 2 Routine. Da die Radix 4 FFT aufgrund der komplexeren Programmstruktur mehr Speicherzugriffe benötigt, ist diese für

große *N* langsamer. Würde mehr „schneller“ Speicher zur Verfügung stehen, ließen sich auch für *N* größer 256 die von *TI* angegebenen Laufzeiten erreichen. Der Abbildung 5.3 ist zu entnehmen, inwieweit die angegebenen Zeiten von den tatsächlich gemessenen Zeiten abweichen

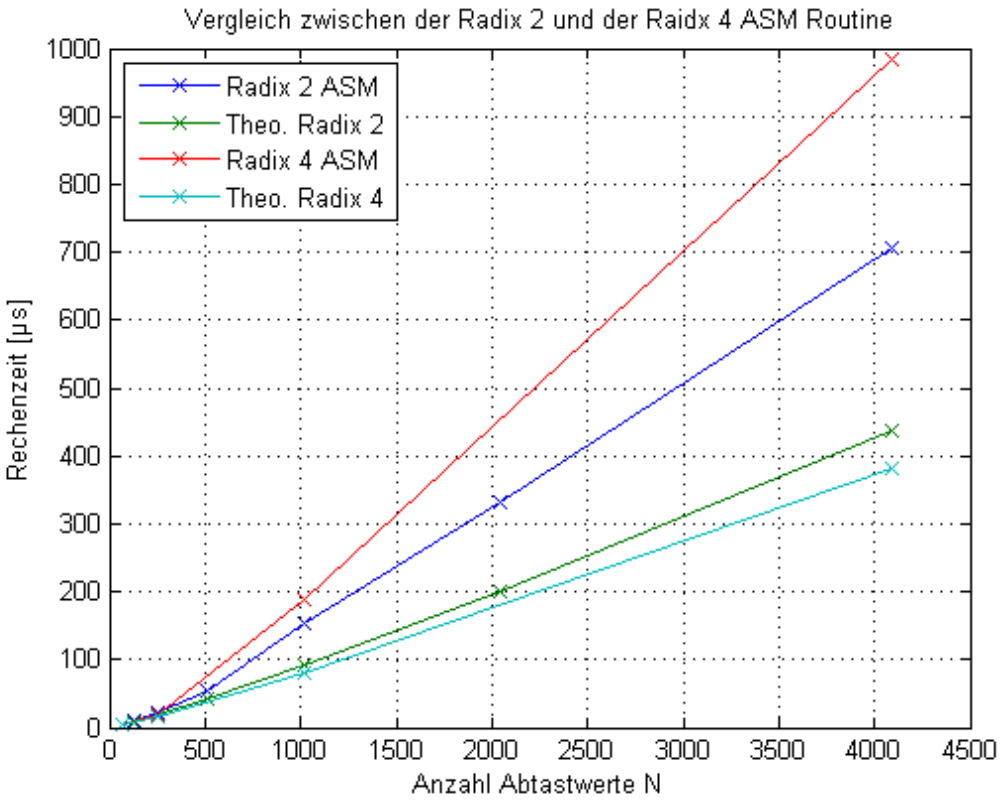


Abbildung 5.3 Vergleich der Radix 2 und Radix 4 ASM Routine

Die längere Rechenzeit der Radix 4 FFT für große *N* widerspricht den theoretischen Untersuchungen (25% weniger komplexe Multiplikationen), diese ist allein auf die Hardwarestruktur zurückzuführen.

Mittels der „*2N* Punkte“ FFT kann eine Radix 4 256 Punkte FFT mit 512 Werten berechnet werden, ohne mehr Speicherplatz für die Berechnung in Anspruch zu nehmen (abgesehen vom Speicherbedarf für 512 Eingangswerte anstatt 256). Eine auf diese Art berechnete „512 Punkte Radix 4 FFT“ benötigt 50,4 µs für die Durchführung und hat damit eine nahezu identische Laufzeit wie die direkte Berechnung einer Radix 2 FFT mit 512 Punkten.

Allgemein bringt die Nutzung der Assembler-Routinen für rechenintensive Programmteile große Geschwindigkeitsvorteile gegenüber dem äquivalenten C-Code. Jedoch hat die Verwendung neben der aufwändigeren Programmstruktur den Nachteil, dass sich diese Codes nicht auf einen anderen Prozessor übertragen lassen, ohne grundlegende Veränderungen am Code vorzunehmen. Dagegen kann ein C-Code mit geringen Änderungen auf ein anderes

System übertragen werden. Die große Laufzeitsteigerung ohne Verlust an Genauigkeit gegenüber einer C-Code Implementierung spricht für die Nutzung der Assembler-Codes.
Durch die Verwendung des EDMA entsteht eine zusätzliche Entlastung der CPU, da diese nicht für das Speichern der Abtastwerte genutzt werden muss. Das Hauptprogramm wird somit nicht unterbrochen und steht für Berechnungen zur Verfügung.

5.2 Resultat

Anders als die theoretischen Überlegungen gezeigt haben, lässt sich durch die Radix 4 FFT auf der Hardware kein Geschwindigkeitsvorteil erzielen. Zudem kann es für spezielle Anwendungen nachteilig sein, dass die Anzahl der Eingangswerte einer Potenz der Basis 4 entsprechen muss.
Die effizienteste Nutzung der Hardware ergibt sich durch die Verwendung der Radix 2 Floating-Point FFT von *Texas Instruments* in Verbindung mit der handoptimierten Routine zum berechnen der Ausgangswerte für *2N* Eingangswerte. Des Weiteren haben die Messungen gezeigt, dass die Konvertierung der Eingangswerte auf den Datentyp „float“, durch die Verwendung der handoptimierten Assembler-Routine einen Geschwindigkeitsvorteil bietet.
Anhand dieser Ergebnisse wurde das Programm „FFT_r2_2N“ erstellt, um mit Hilfe der folgenden Funktionen eine *N* Punkte FFT effizient zu berechnen. Genutzt werden dafür folgende Dateien:

- int_to_float.asm zum Maskieren und Konvertieren der Eingangswerte
- cfftr2_dit.asm zur Berechnung der FFT
- bitrev.asm zum Sortieren der Ausgangswerte in natürlicher Reichenfolge
- calc_piped.asm zur Berechnung der 2N Werte
- sum.asm zur Berechnung des Betragsspektrums

Durch die Verwendung dieser Funktionen ergeben sich die in

Tabelle 5.4 dargestellten Laufzeiten. Wobei in der linken Spalte die mittels Simulator ermittelte Anzahl an benötigten Takten und die daraus resultierende, theoretische Laufzeit dargestellt ist. Der rechten Spalte sind die mittels Hardware-Profiling ermittelten Laufzeiten zu entnehmen.

Tabelle 5.4 Laufzeiten für das Programm FFT_r2_2N

2*N	Cycles	Theo. Zeit[µs]	Zeit [µs]
128	2635	11.71	12.0
256	4436	19.72	20.0
512	10923	48.55	50.7
1024	25689	114.17	114.8
2048	63178	280.79	282
4096	134272	596.76	598

Die Umsetzung der rechenintensiven Programmteile in Assembler hat gezeigt, dass sich die Rechenzeit im Vergleich zu einem in C geschriebenen Programm wesentlich verkürzen lässt.

5.2.1 Programmbeschreibung

Die oben genannten Programmteile wurden zusammen mit den für die Nutzung des DSPs notwendigen Programmen in einem Projekt zusammengefasst. Folgende Dateien enthalten sind in diesem Projekt enthalten:

- bitrev.asm
- C6713dsk_EDMA.cmd
- c6713dskinit_EDMA.c
- calc_piped.asm
- cfftr2_dit.asm
- digitrev_index.c
- int_to_float_piped.asm
- main_FFT_r2_2N.c
- sum.asm
- Vectors_intr.asm

Dieses Projekt kann einfach durch das Aufrufen der gleichnamigen „Workspace“-Datei in Code Composer Studio geöffnet werden.

Nach dem Start des Programms werden alle nötigen Initialisierungen vorgenommen. Dazu gehören die Initialisierung des DSK-Bords und der nötigen Variablen sowie die Konfiguration des EDMA und die Berechnung der Twiddle-Faktoren. Nach dem Start des EDMA beginnt dieser mit dem Aufnehmen der Abtastwerte.
Stehen der CPU *2N* neue Abtastwerte zur Verfügung, werden diese mit der Funktion „int_to_float( )“ auf den Datentyp „float“ konvertiert. Daraufhin wird die FFT berechnet und anschließend werden die Ausgangswerte durch die Funktion „bitrev( )“ in natürlicher Reihenfolge sortiert. Zum Schluss werden die *2N* Punkte für das Spektrum berechnet und für eine

Weiterverarbeitung in das Array „out_buf“ gespeichert. Dieser Programmablauf ist in Abbildung 5.4 dargestellt.

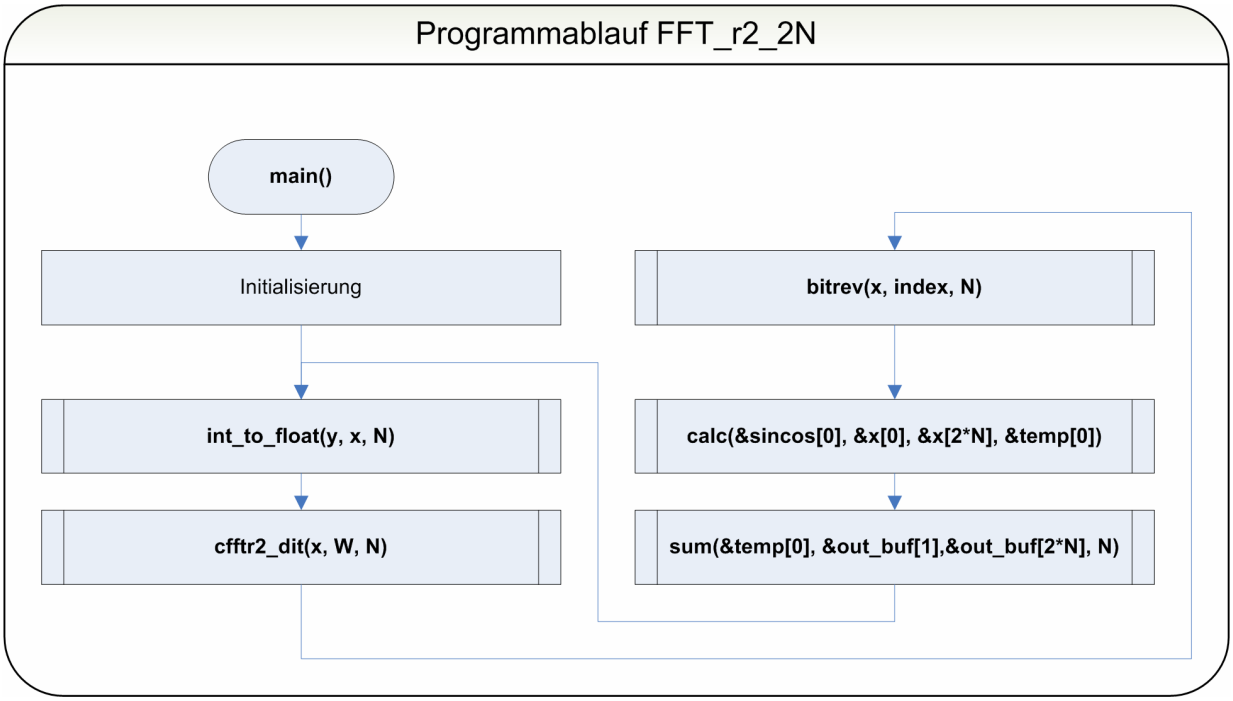


Abbildung 5.4 Programmablauf fft_r2_2N

Eine Beschreibung der Funktionen mit Prototypen und Übergabeparametern befindet sich in Anhang E.

Einstellbare Parameter

Um das Programm möglichst flexibel einsetzen zu können, können die Abtastfrequenz, die Anzahl der Eingangswerte und wahlweise eine Gewichtung der Eingangswerte mit einem Hamming-Fenster im Kopf des Programms „main_FFT_r2_2N.c“, eingestellt werden.
Die Abtastfrequenz f_s lässt sich einfach über die in Tabelle 2.2 dargestellten Makros einstellen. Ebenso lässt sich die Anzahl der Punkte für die FFT Berechnung über das Präprozessor Makro *N* einstellen. Dabei ist zu beachten, dass $2 \cdot N$ Punkte mittels der *N* Punkte FFT berechnet werden (Beispiel: für *N*=256 wird das Spektrum von 512 Abtastwerten berechnet). Des Weiteren ist zu beachten, dass *N* ebenfalls in der Datei „calc_piped.asm“ als Hexadezimalwert eingetragen werden muss (der Quellcode ist an den entsprechenden Stellen kommentiert).
Über das Makro „WINDOW“ kann wahlweise eine Gewichtung der Eingangswerte mit einem Hamming-Fenster eingestellt werden.
Im Programm „int_to_float.asm“ lässt sich einstellen, welcher der beiden Kanäle für die Berechnung verwendet werden soll. Der Programmcode ist an den entsprechenden Stellen kommentiert.

6 Schlussbemerkung

6.1 Fazit

Durch die sukzessive Optimierung der einzelnen Programmabschnitte wurde das Programm „FFT_r2_2N“ entwickelt. Dieses Programm umfasst neben der FFT Funktion auch eine effiziente Vor- und Nachbereitung der Daten. Durch die erzielten Zeiteinsparungen des Programms steht im Vergleich zu anderen FFT-Berechnungsprogrammen mehr Rechenzeit für die eigentlichen Berechnungen zur Verfügung, wie z.B. für die Berechnung einer schnellen Faltung.
Nach Meinung des Verfassers sind die zur Berechnung nötigen Funktionen hinreichend gut optimiert, bis auf die Funktion „sum.asm“. Bei dieser Funktion ließe sich durch Sorftware-Pipelining die Laufzeit noch weiter reduzieren. Dieses ist bisher jedoch noch nicht vollständig gelungen.
Trotz der verbleibenden Optimierungsmöglichkeiten wurde das Ziel erreicht, einen effizienten Code zur Berechnung einer schnellen Spektralanalyse auf dem digitalen Signalprozessor Bord „TMS320C6713“ zu implementieren.

6.2 Ausblick

Eine weitere Möglichkeit, um die Rechengeschwindigkeit weiter zu erhöhen, wäre die Implementierung einer Radix 8 FFT in Assembler, da diese theoretisch 41% weniger komplexe Multiplikationen im Vergleich zur Radix2 FFT benötigt. Dies ist nach Meinung des Verfassers auf dem „TMS320C6713“ nicht sinnvoll, weil die Programmstruktur zur Berechnung der Radix 8 FFT im Vergleich zur Radix 2 FFT komplexer ist. Wie die Untersuchungen gezeigt haben, ist schon bei der Radix 4 FFT zu erkennen, dass für eine direkte (und somit schnelle) Berechnung nicht genügend Registerpaare zur Verfügung stehen. Jedoch könnte die Implementierung einer Radix 8 FFT auf einem anderen Prozessor, wie z.B. dem „TMS320C6416“, einen Geschwindigkeitsvorteil bringen, da hier 2×32 Register anstatt 2×16 Register für die Berechnung zur Verfügung stehen.
Eine geringfügige Verbesserung würde sich durch das „Pipelinen“ der Funktion „sum.asm“ ergeben. Eine solche Version des Programms konnte aus Zeitgründen nicht fertig gestellt werden.
Davon abgesehen bietet die Berechnung der FFT auf dieser Hardware nach Meinung des Verfassers wenig Potential für weitere Verbesserungen.

7 Literaturverzeichnis

[1] Born 1959 BORN, M., WOLF, M.: Principles of Optics. New York: Pergamon Press, 1959

[2] Bracewell 1961 BRACEWELL, Ron: The Fourier Transform and Its Applications. New York: McGraw-Hill, 1965

[3] Brigham 1995 BRIGHAM, Elbert Oran: Schnelle Fourier-Transformation, 6. Auflage. München Wien: Oldenbourg, 1995

[4] Brügmann 2009 BRÜGMANN, Marco: Hardwarenahe Implementierung eines Bildverarbeitungsalgorithmus auf einem DSP-Board zur Gesichtsdetektion. Hamburg, 2009

[5] Cassaing 2005 CHASSAING, Rulph: Digital Signal Processing and Applications with the C6713 and C6416 DSK. New Jersey: J. Wiley, 2005

[6] Dietrich 2009 DIETRICH, Sebastian: Teile und herrsche. Wikipedia, Stand: 13.11.2009.URL:http://de.wikipedia.org/wiki/Teile_und_herrsche. (Informatik). (abgerufen am 15.01.2010)

[7] Douglas 2010 DOUGLAS L. Jones: Radix-4 FFT Algorithms: Connexions, Stand 10.01.2010. URL : http://cnx.org/content/m12027/

[8] Gupta 1966 GUPTA, S.C.: Transform and State Variable Methods in Linear Systems. New York: Wiley, 1966

[9] Jones 2006 JONES, Douglas: Split-Radix FFT Algorithms. US/Central 2006

[10] Kölzer 2006 KÖLZER, Hans Peter, REICHARDT, Jürgen: Fast Fourier Transform (FFT), 2006

[11] Kraus 1950 KRAUS, J.O.: Antennas. New York: McGraw-Hill, 1950

[12] Lee 1960 LEE, Y.W.: Statistical Theory of Communikation. New York: Wiley, 1960

[13] Müller-Wichards 1999 MÜLLER-WICHARDS, Dieter: Transformationen und Signale. Leiptzig: Teubner 1999

[14] Oppenheim 1992 OPPENHEIM, Alan, SCHAFER Ronald: Zeitdiskrete Signalverarbeitung. München Wien: Oldenbourg, 1992

[15] Proakis 2007 PROAKIS, John, MANULAKIS, Dimitris: Digital Signal Processing. Person Prentice Hall 2007

[16] Sauvagerd 2009 SAUVAGERD, Ulrich: Radix-2 FFT for real sequences. Hamburg 2009

[17] Scholz 2001 SCHOLZ, Dieter: Diplomarbeit normgerecht verfassen. Würzburg: Vogel, 2001

[18] Spra291 2001 TEXAS INSTRUMENTS INC.: Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS320 DSP Platform, 2001

[19] Spra636a 2001 TEXAS INSTRUMENTS INC.: Applications Using the TMS320C6000 Enhanced DMA, 2001

[20] Spru187o 2008 TEXAS INSTRUMENTS INC.: TMS320C6000 Optimizing Compiler v 6.1, 2008

[21] Spru198g 2002 TEXAS INSTRUMENTS INC.: TMS320C6000 Programmers´s Guide, 2002

[22] Spru234c 2006 TEXAS INSTRUMENTS INC.: TMS320C6000 DSP Enhanced Direct Memory Access (EDMA) Controller Reference Guide, 2006

[23] Spru580g 2006 TEXAS INSTRUMENTS INC.: TMS320C6000 DSP Multichannel Buffered Serial Port (McBSP) Reference Guide, 2006

[24] Spru733a 2008 TEXAS INSTRUMENTS INC.: TMS320C67x/67x+ DSP CPU and Instruction Set Reference Guide, 2008

[25] TLV320AIC23 2002 TEXAS INSTRUMENTS INC.: TLVAIC23 Stereo Audio Codec, Data Manual, 2002

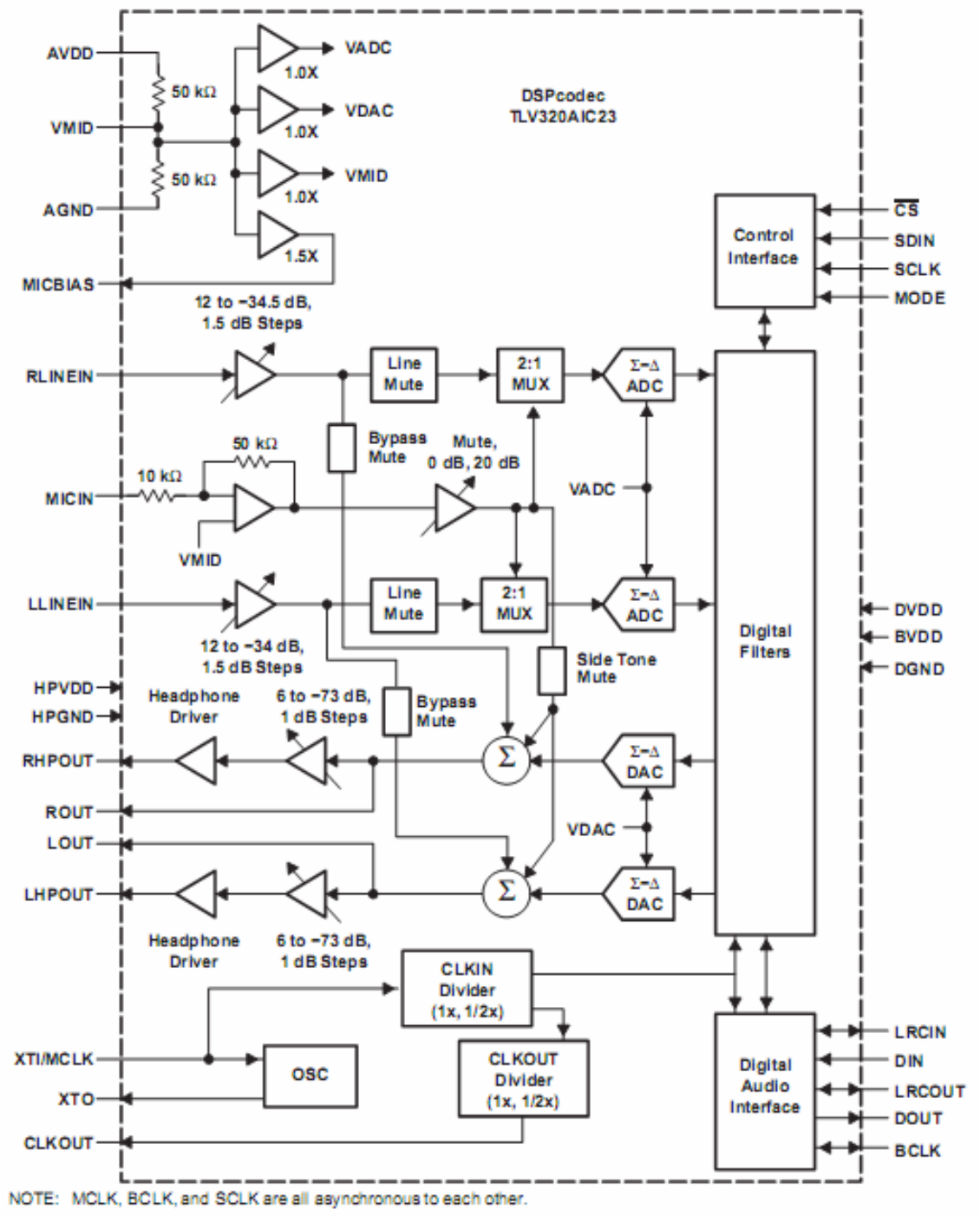
[26] TMS320C6713b 2006 TEXAS INSTRUMENTS INC.: Floting-Point Digital Signal Processor, 2006

[27] TMS320C6713 2003 SPEKTRUM DIGITAL INC.: TMS320C6713 DSK Technical Reference, 2003

[28] Tretter 2008 TRETTER, A. Steven: Communication System Design Using DSP Algorithms. Maryland: Springer, 2008

Anhang

A Blockdiagramm des TLVAIC23 Codec



B Assemblerbefehle für Fix- und Floating-Point -Operationen

.L Unit	.M Unit	.S Unit	.D Unit
ABS	MPY	ADD	ADD
ADD	MPYH	ADDK	ADDAB
ADDU	MPYHL	ADD2	ADDAH
AND	MPYHLU	AND	ADDAW
CMPEQ	MPYHSLU	B disp	LOB
CMPGT	MPYHSU	B IRP[a]	LDBU
CMPGTU	MPYHU	B NRP[a]	LDH
CMPLT	MPYHULS	B reg	LDHU
CMPLTU	MPYHUS	CLR	LDW
LMBD	MPYLH	EXT	LOB (15-bit offset)[b]
MV	MPYLHU	EXTU	LDBU (15-bit offset)[b]
NEG	MPYLSHU	MV	LDH (15-bit offset)[b]
NORM	MPYLUHS	MVC[a]	LDHU (15-bit offset)[b]
NOT	MPYSU	MVK	LDW (15-bit offset)[b]
OR	MPYU	MVKH	MV
SADD	MPYUS	MVKLH	STB
SAT	SMPY	NEC	STH
SSUB	SMPYH	NOT	STW
SUB	SMPYHL	OR	STB (15-bit offset)[b]
SUBU	SMPYLH	SET	STH (15-bit oftset)[b]
SUBC		SHL	STW (15-bit offset)[b]
XOR		SHR	SUB
ZERO		SMRU	SUBAB
		SSHL	SUBAH
		SUB	SUBAW
		SUBU	ZERO
		SUB2	
		XOR	
		ZERO	

[a]nur in S2 [b]nur in D2

C Assemblerbefehle für Floating-Point Operationen

.L Unit	.M Unit	.S Unit	.D Unit
ADDDP	MPYDP	ABSDP	ADDAD
ADDSP	MPYI	ABSSP	LDDW
DPINT	MPYID	CMPEQDP	
DPSP	MPYSP	CMPEQSP	
DPTRUNC		CMPGTDP	
INTDP		CMPGTSP	
INTDPU		CMPLTDP	
INTSP		CMPLTSP	
INTSPU		RCPDP	
SPINT		RCPSP	
SPTRUNC		RSQRDP	
SUBDP		RSQRSP	
SUBSP		SPDP	

D Programmablaufplan calc_piped.asm

Takt	1	2	3	4	5	6
A						
D1	LDDW A15:A14	LDDW B13:B12				
L1	MVKL A1	MVKLH A1				
S1	MVKL A13	MVKLH A13				
M1						MPYSP A13,B14,A11
B						
D2	LDDW B15:B14					
L2				MV 13,B0		
S2						
M2						MPYSP B15,B0,B11
Start of Iteration	1					
End of Iteration						
	1					
		2				6
Branch						
Brachn Execution						

7	8	9	10	11	12	13
LDDW A15:A14	LDDW B13:B12					
				ADDSP A11,A10,A9		LDDW A15:A14
MPYSP A13,B12,A10	MPYSP A15,B15,A11	MPYSP A15,B13,A10	MPYSP A14,B12,A11	MPYSP A14,B14,A10	MPYSP A13,B14,A11	MPYSP A13,B12,A10
LDDW B15:B14						
				SUBSP B11,B10,B9		LDDW B15:B14
MPYSP B13,B0,B10	MPYSP B12,A15,B11	MPYSP B14,A15,B10	MPYSP B15,A14,B11	MPYSP B13,A14,B10	MPYSP B15,B0,B11	MPYSP B13,B0,B10
2						3
1						1
	2					
					6	
7						7
	8					
		9				
			10			
				11		
						13

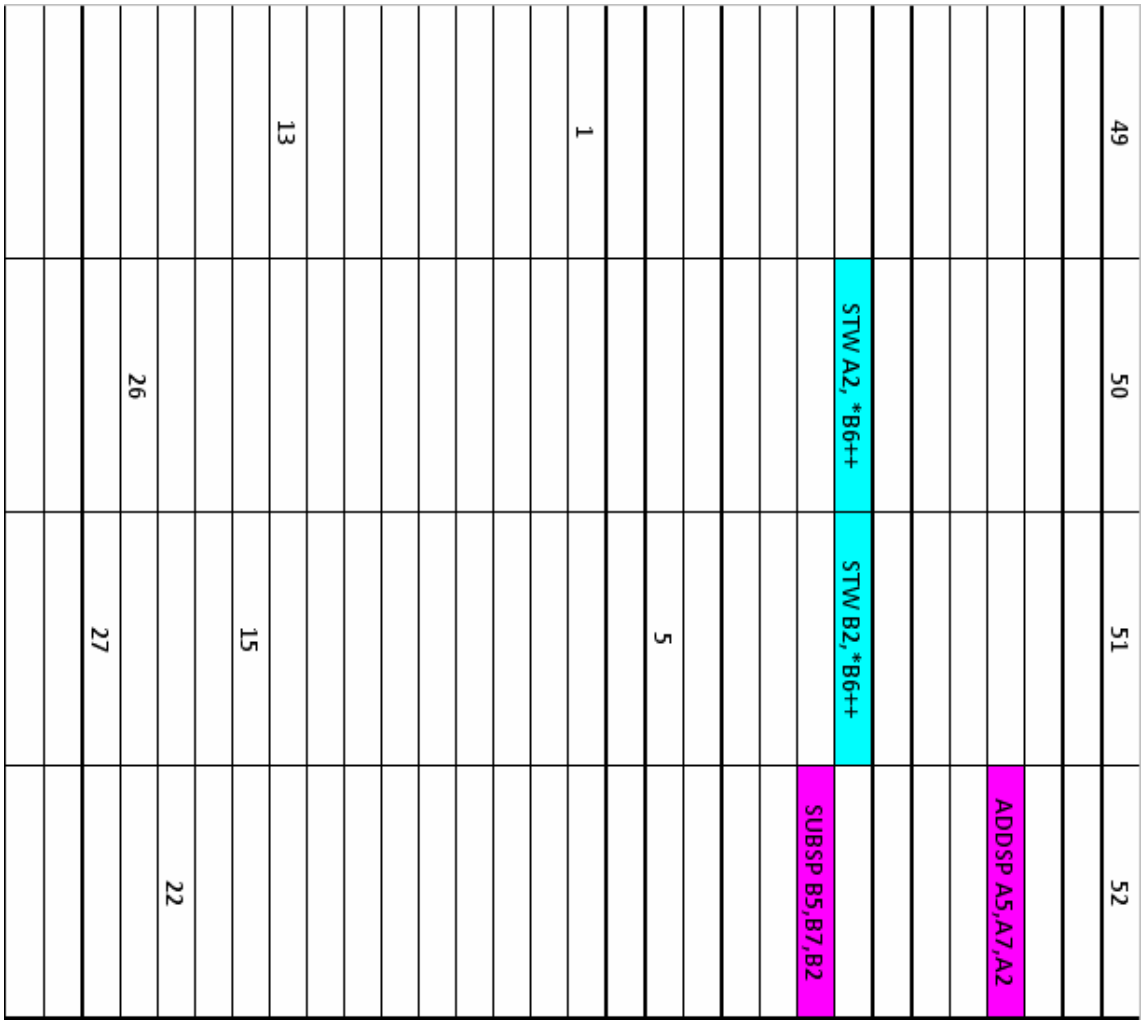
14	15	16	17	18	19	20
LDDW B13:B12					LDDW A15:A14	LDDW B13:B12
	SUBSP A11,A10,A7		ADDSP A11,A10,A9	ADDSP A9,A8,A5	ADDSP A11,A10,A8	
MPYSP A15,B15,A11	MPYSP A15,B13,A10	MPYSP A14,B12,A11	MPYSP A14,B14,A10	MPYSP A13,B14,A11	MPYSP A13,B12,A10	MPYSP A15,B15,A11
					LDDW B15:B14	
	ADDSP B11,B10,B7		SUBSP B11,B10,B9	ADDSP B9,B8,B5	SUBSP B11,B10,B8	
MPYSP B12,A15,B11	MPYSP B14,A15,B10	MPYSP B15,A14,B11	MPYSP B13,A14,B10	MPYSP B15,B0,B11	MPYSP B13,B0,B10	MPYSP B12,A15,B11
					4	
					1	
2				6		2
					7	
8						8
	9					
		10				
			11		13	
	15			18		

21	22	23	24	25	26	27
				LDDW A15:A14	LDDW B13:B12	
SUBSP A11,A10,A7	ADDSP A5,A7,A2	ADDSP A11,A10,A9	ADDSP A9,A8,A5	ADDSP A11,A10,A8		SUBSP A11,A10,A7
MPYSP A15,B13,A10	MPYSP A14,B12,A11	MPYSP A14,B14,A10	MPYSP A13,B14,A11	MPYSP A13,B12,A10	MPYSP A15,B15,A11	MPYSP A15,B13,A10
ADDSP B11,B10,B7	SUBSP B5,B7,B2	SUBSP B11,B10,B9	ADDSP B9,B8,B5	LDDW B15:B14	STW A2, *B6++	STW B2,*B6++
MPYSP B14,A15,B10	MPYSP B15,A14,B11	MPYSP B13,A14,B10	MPYSP B15,B0,B11	SUBSP B11,B10,B8		ADDSP B11,B10,B7
				B		
				5		1
				1		
			6		2	
				7	8	9
9						
	10	11		MPYSP B13,B0,B10	MPYSP B12,A15,B11	MPYSP B14,A15,B10
				13		15
15						
			18			
	22				26	
				b		27

28	29	30	31	32	33	34
ADDSP A5,A7,A2	ADDSP A11,A10,A9	ADDSP A9,A8,A5	LDDW A15:A14	LDDW B13:B12		
			ADDSP A11,A10,A8		SUBSP A11,A10,A7	ADDSP A5,A7,A2
MPYSP A14,B12,A11	MPYSP A14,B14,A10	MPYSP A13,B14,A11	MPYSP A13,B12,A10	MPYSP A15,B15,A11	MPYSP A15,B13,A10	MPYSP A14,B12,A11
SUBSP B5,B7,B2	SUBSP B11,B10,B9	ADDSP B9,B8,B5	LDDW B15:B14	STW A2, *B6++	STW B2, *B6++	
			SUBSP B11,B10,B8		ADDSP B11,B10,B7	SUBSP B5,B7,B2
MPYSP B15,A14,B11	MPYSP B13,A14,B10	MPYSP B15,B0,B11	MPYSP B13,B0,B10	MPYSP B12,A15,B11	MPYSP B14,A15,B10	MPYSP B15,A14,B11
			6		2	
			1			
				2		
		6				
			7			
				8		
					9	
10						10
	11					
			13			
					15	
		18				
22						22
				26		
					27	
		b				

35	36	37	38	39	40	41
ADDSP A11,A10,A9	ADDSP A9,A8,A5	ADDSP A11,A10,A8		SUBSP A11,A10,A7	ADDSP A5,A7,A2	ADDSP A11,A10,A9
MPYSP A14,B14,A10	MPYSP A13,B14,A11	MPYSP A13,B12,A10	MPYSP A15,B15,A11	MPYSP A15,B13,A10	MPYSP A14,B12,A11	MPYSP A14,B14,A10
			STW A2, *B6++	STW B2,*B6++		
SUBSP B11,B10,B9	ADDSP B9,B8,B5	SUBSP B11,B10,B8		ADDSP B11,B10,B7	SUBSP B5,B7,B2	SUBSP B11,B10,B9
MPYSP B13,A14,B10	MPYSP B15,B0,B11	MPYSP B13,B0,B10	MPYSP B12,A15,B11	MPYSP B14,A15,B10	MPYSP B15,A14,B11	MPYSP B13,A14,B10
		7		3		
		1				
			2			
	6					
		7				
			8			
				9		
					10	
11						11
		13				
				15		
	18					
					22	
			26			
				27		

42	43	44	45	46	47	48
ADDSP A9,A8,A5	ADDSP A11,A10,A8		SUBSP A11,A10,A7	ADDSP A5,A7,A2		ADDSP A9,A8,A5
		STW A2, *B6++	STW B2,*B6++			
ADDSP B9,B8,B5	SUBSP B11,B10,B8		ADDSP B11,B10,B7	SUBSP B5,B7,B2		ADDSP B9,B8,B5
			4			
	1					
		2				
6						
	7					
		8				
			9			
				10		
					11	
	13					
			15			
18						18
				22		
		26				
			27			


49
50
51
52
ADDSP A5,A7,A2
STW A2, *B6++
STW B2, *B6++
SUBSP B5,B7,B2
1
5
13
15
22
26
27

E Funktionsbeschreibungen

Konvertieren der Daten von „integer“ zu „float“:

Funktion int_to_float()	
Prototyp:	int int_to_float(int*, float*, int)
Beschreibung:	Konvertierung eines Array der Länge N von "int" zu "float"
Parameter:	Array mit integerwerten, Array mit Float-Werten, Länge der Arrays
Datei:	int_to_float.asm

Berechnung der FFT mit *N* Komplexen Eingangswerten:

Funktion cfftr2_dit()	
Prototyp:	void cfftr2_dit(float[2N], float[3N/2], int)
Beschreibung:	Berechnung der FFT mit N Komplexen Werten
Parameter:	Array mit Eingangswerten, Array Twiddle-Faktoren, Länge N des Arrays
Datei:	cfftr2_dit.asm

Sortieren der Daten in natürliche Reichenfolge:

Funktion bitrev()	
Prototyp:	void bitrev(float*, short*, int)
Beschreibung:	Sortieren der Koeffizienten in natürliche Reihenfolge
Parameter:	Array mit Eingangswerten, Array der Indexelemente, Länge N des Arrays
Datei:	bitrev.asm

Berechnung der *2N* Werte aus der *N* Punkte FFT:

Funktion calc()	
Prototyp:	void calc(float*, float*, float*, float*)
Beschrei-bung:	Berechnung der 2N Ausgangswerte
Parameter:	Pointer auf Tabelle mit sin und cos Werten, Pointer auf erstes und letztes Element der Eingangswerte, Array zum speichern der Zwischenergebnisse
Datei:	calc_piped.asm

Bilden des Betragsspektrums:

Funktion sum();	
Prototyp:	void sum(float*, float*, float*, int)
Beschreibung:	Berechnung des Betragsspektrums
Parameter:	Pointer auf Zwischenspeicher, Pointer auf erstes und letztes Element des Ergebnisvektors, Länge N des Array
Datei:	sum.asm

F Quellcode

Der Quellcode befindet sich auf der beiliegenden CD. Neben dem Programm „FFT_r2_2N“ sind auch alle anderen in dieser Arbeit beschriebenen Programme enthalten. Die CD kann bei Herrn Prof. Dr.-Ing. Kölzer eingesehen werden.

Danksagung

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. -Ing. Hans Peter Kölzer, meinem betreuenden Prüfer, dafür bedanken, dass er es mir ermöglicht hat, diese Bachelorthesis zu erstellen, und mich ebenso durch interessante Anregungen unterstützt hat. Des Weiteren bedanke ich mich bei Herrn Prof. Dr. -Ing. Ulrich Sauvagerd für seine Arbeit als Zweitprüfer und für seine nützlichen Hinweise zur Programmierung.

Ein besonderer Dank geht zum einen an meine Eltern, Sigrid und Heinrich Dieckmann, die mir dieses Studium erst ermöglicht haben, und zum anderen an meine Freundin, Christiane Jansing, die mich während meines gesamten Studiums sehr unterstützt hat.

Für das Korrekturlesen dieser Arbeit bedanke ich mich bei meiner Tante Anneliese Dieckmann.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig 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, den 12.02.2010

Ort, Datum

Unterschrift