

Masterarbeit

Abdulhai Al-Mohamed Zein

Hardware-Software-Codesign eines OFDM-Systems auf
einer DSP-FPGA-Plattform unter Verwendung von
High-Level-Synthese

Abdulhai Al-Mohamed Zein

Hardware-Software-Codesign eines OFDM-Systems auf einer DSP-FPGA-Plattform unter Verwendung von High-Level-Synthese

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Masterstudiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Fakultät Elektro-, Medien- und Informationstechnik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Lutz Leutelt

Zweitgutachter: Prof. Dr.-Ing. Sönke Appel

Eingereicht am: 05. Dezember 2025

Abdulhai Al-Mohamed Zein

Thema der Arbeit

Hardware-Software-Codesign eines OFDM-Systems auf einer DSP-FPGA-Plattform unter Verwendung von High-Level-Synthese

Stichworte

OFDM, 16-QAM, FFT/IFFT, Cyclic Prefix (zyklisches Präfix), Signal-Rausch-Verhältnis (SNR), DSP-FPGA Codesign, High-Level-Synthese (HLS), C/C++, VHDL, EMIFA, EDMA, AXI4-Stream.

Kurzzusammenfassung

Diese Arbeit entwirft und realisiert ein Orthogonal Frequency Division Multiplexing (OFDM)-System im Rahmen eines Hardware-Software-Codesigns auf einer heterogenen DSP-FPGA-Plattform. Ausgehend von einer profilbasierten DSP-Baseline werden performanzkritische Verarbeitungsschritte – insbesondere FFT/IFFT, QAM-(De)Mapping und Cyclic Prefix (CP)-Verarbeitung – identifiziert und mithilfe von High-Level-Synthese (HLS) in dedizierte FPGA-IP-Module ausgelagert. Die Steuer- und Protokolllogik verbleibt auf dem DSP. Die Datenkopplung erfolgt speichergemappt über die External Memory Interface A (EMIFA)-Schnittstelle, innerhalb des FPGAs wird AXI4-Stream zur internen Verbindung verwendet. Die Leistungsbewertung umfasst Messungen zu Durchsatz, Latenz und Ressourcennutzung (LUTs, Flip-Flops, BRAMs, DSP-Slices).

Abdulhai Al-Mohamed Zein

Title of Thesis

Hardware-software codesign of an OFDM system on a DSP-FPGA platform using high-level synthesis

Keywords

OFDM, 16-QAM, FFT/IFFT, Cyclic Prefix, Signal-to-Noise Ratio (SNR), DSP-FPGA Codesign, High-Level Synthesis (HLS), C/C++, VHDL, EMIFA, EDMA, AXI4-Stream.

Abstract

This thesis designs and implements an Orthogonal Frequency Division Multiplexing (OFDM) system within the framework of hardware-software codesign on a heterogeneous DSP-FPGA platform. Based on a profile-driven DSP baseline, performance-critical processing stages—particularly FFT/IFFT, QAM (de)mapping, and cyclic prefix (CP) handling—are identified and offloaded to dedicated FPGA IP modules using High-Level Synthesis (HLS). Control and protocol logic remain on the DSP. Data exchange is performed via a memory-mapped External Memory Interface A (EMIFA) interface, while AXI4-Stream is employed internally on the FPGA for high-throughput interconnection. The system evaluation includes measurements of throughput, latency, and resource utilization (LUTs, flip-flops, BRAMs, and DSP slices).

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

Danksagung

الشُّكْرُ وَالتَّقْدِيرُ

An erster Stelle gilt mein tiefster Dank meinen Eltern und meinen Geschwistern **Reem, Bayan, Mona, Khaled und Ahmad**. Ihnen widme ich diese Arbeit aus tiefstem Herzen. Meine Mutter und mein Vater sind die Menschen, die mir gezeigt haben, was wahre Stärke, Geduld und Liebe bedeuten. Sie haben an mich geglaubt, selbst dann, wenn ich selbst gezweifelt habe, und sie waren mein sicherer Halt in jeder Lebensphase. Ohne sie wäre dieser Weg nicht möglich gewesen. Ein besonderer Dank geht an meine Freundin **Thi Dung Nguyen**, die mich in allen Höhen und Tiefen begleitet hat — mit Verständnis, Geduld und einem Herzen voller Unterstützung. Mein aufrichtiger Dank gilt auch meinem Onkel **Erk Yontar** und seiner Familie, die immer an mich geglaubt und mich motiviert haben, nicht aufzugeben. Abschließend danke ich meinem Betreuer **Prof. Dr. Lutz Leutelt**, der mir mit Rat, Geduld und Vertrauen zur Seite stand.

في المقام الأول، أقدم خالص الشكر والامتنان لوالدي العزيزين ولإخوتي الأحبّة - ريم، بيان، موني، خالد، وأحمد - وأهدي إليهم هذا العمل من أعماق قلبي. لقد كان والدي مثالاً للصبر والعطاء، دعماي دون حدود، وأمتنا بي حتى في لحظات ضعفي وشكي. لولاهما لما وصلت إلى ما أنا عليه اليوم، فهما منبع الحب والقوة والإلهام في حياتي. كما أعبر عن امتناني العميق لصديقتي العزيزة Thi Dung Nguyen التي كانت إلى جانبي في كل خطوة، في الأيام الجميلة والصعبة على حد سواء، ولم تبخل يوماً بدعائها وتشجيعها الصادق. ولا أنسى أن أقدم جزيل الشكر لـ Erk Yontar وعائلته الكريمة الذين لم يتوقفوا يوماً عن الإيمان بي وتخفيفي على المضي قدماً. وأخيراً، أوجه خالص التقدير والعرفان للأستاذ Prof. Dr. Lutz Leutelt على توجيهاته ودعاه وثقته التي كان لها أثر بالغ في إتمام هذا العمل.

Abdulhai Al-Mohamed Zein



Inhaltsverzeichnis

Abbildungsverzeichnis	x
Tabellenverzeichnis	xiii
Listings	xiv
1 Einleitung	1
1.1 Einführung und Motivation	1
1.2 Ziele der Arbeit	2
1.3 Gliederung der Arbeit	2
2 Grundlagen	3
2.1 Orthogonal Frequency-Division Multiplexing (OFDM)	3
2.1.1 Mathematische Grundlagen	4
2.1.2 OFDM-Sendekette und Frameaufbau	13
2.1.3 OFDM-Empfängerkette	19
2.1.4 Peak-to-Average Power Ratio (PAPR)	24
2.2 Faltungscodes	25
2.2.1 Grundidee und Einsatz	25
2.2.2 Encoderprinzip	26
2.2.3 Decodierung	27
2.3 High-Level-Synthese (HLS)	28
2.3.1 Grundlagen der High-Level-Synthese	28
2.3.2 Werkzeuge und Design-Flow	29
2.3.3 Pragmas und Optimierungsdirektiven in HLS	30
2.4 Hardwareplattform	32
2.4.1 DSP-Board TMS320C6747	32
2.4.2 FPGA-Board: Artix-7 XC7A100T	33

2.5	Asynchrones Bussystem EMIFA	34
2.5.1	Signale, Timing und Betriebsarten	35
2.5.2	Rolle des EMIFA im DSP-FPGA-Codesign	37
2.5.3	EDMA im DSP	37
2.6	AXI4-Stream Bus	38
2.6.1	Funktionsweise und Signalübersicht	38
3	Anforderungsanalyse	40
3.1	Einleitung	40
3.2	Ziele und Vorgehen	40
3.3	Funktionale Anforderungen	43
3.4	Nicht-Funktionale Anforderungen	45
4	Konzept und Vorgehensweise	47
4.1	Einleitung	47
4.2	Simulationsmodell	47
4.3	Zielarchitektur des OFDM-Systems	56
4.3.1	Blockdiagramm und Partition	56
4.4	Kommunikationsarchitektur DSP-FPGA über EMIFA	60
4.4.1	Datenübertragung über EMIFA: Wort- vs. Blockweise	60
4.4.2	EMIFA-Empfangsmodul auf FPGA-Seite	61
4.4.3	EMIFA-Interface-Taktung: asynchron vs. synchron	62
4.4.4	Datenpfade und Pufferkonzept	64
4.4.5	Register-Map statt GPIO-Steuerung	65
4.5	Datenpfad-Integration: FIFO ↔ AXI4-Stream	66
4.6	HLS-IP-Kette: Auswahl und Architekturprinzipien	68
4.6.1	Funktionsumfang (erstes Inkrement)	68
4.6.2	HLS-Entwurfsprinzipien zur Taktoptimierung	70
4.7	Systemintegration und Duplex-Fähigkeit	71
5	Implementierung des OFDM-Systems	74
5.1	Überblick und Zielsetzung	74
5.2	Entwicklungsumgebung	74
5.2.1	Toolchain, Plattform und Versionen	75
5.3	DSP-seitige Implementierung	75
5.3.1	EMIFA-Konfiguration	76
5.3.2	EDMA-Konfiguration	79

5.3.3	Pufferung und Datenpfad	82
5.4	FPGA-seitige Systemintegration	86
5.4.1	EMIF-Interface und FIFO-Anbindung	86
5.4.2	AXI4-Stream: FIFO $\leftrightarrow$ AXI	89
5.4.3	Integration der Funktionsmodule	94
5.5	High-Level-Synthese der Verarbeitungskette	97
5.5.1	Schnittstellen und Steuerung	97
5.5.2	Signalverarbeitungsblöcke (16QAM, IFFT, CP)	99
5.5.3	HLS-basierte Ressourcenschätzung für 16-Bit-Twiddle-Faktoren . .	107
5.6	Sender- und Empfängerseitige Verarbeitung	108
5.6.1	Senderkette (TX)	108
5.6.2	Empfängerkette (RX)	109
5.7	Vergleichsbasis DSP-Only	111
5.8	Vorwärtsfehlerkorrektur (FEC)	112
5.8.1	Faltungscodes: Encoder und Decoder	112
5.9	Verifikation und Co-Simulation	113
5.9.1	Gesamtsystemverifikation	113
6	Ergebnisse	116
6.1	Funktionale Verifikation	116
6.1.1	Integration der Vorwärtsfehlerkorrektur (FEC)	116
6.1.2	Audioverifikation im Echtzeitbetrieb	117
6.2	Leistungsbewertung und Vergleich	119
6.2.1	Durchsatz und Latenz	119
6.2.2	Ressourcenauslastung auf dem FPGA	122
6.2.3	Syntheseergebnisse in Vivado	123
6.3	Zusammenfassung der Ergebnisse	124
7	Zusammenfassung und Ausblick	126
7.1	Systembewertung und Reflexion	126
7.2	Ausblick	127
	Literaturverzeichnis	128
A	Anhang	135
A.1	BER vergleich für $N_{\text{FFT}} \in \{64, 256\}$	138
A.2	Vergleich der Subträgerlinien für $N_{\text{FFT}} \in \{64, 256\}$	139

A.3 EMIFA Messungen Schreib- Lesevorgang	140
Selbstständigkeitserklärung	141

Abbildungsverzeichnis

2.1	Betragsspektrum von fünf Subträgern bei klassischem FDM: Keine Orthogonalität, spektrales Übersprechen deutlich sichtbar.	5
2.2	Betragsspektrum von fünf orthogonalen OFDM-Subträgern: Nullstellen der Nachbarn exakt getroffen – keine Interferenz trotz spektraler Überlappung.	6
2.3	Einflüsse auf die Ausbreitung der elektromagnetischen Wellen im realen Mobilfunkkanal [31].	10
2.4	Zeitbereichsdarstellung zweier aufeinanderfolgender OFDM-Symbole mit Cyclic Prefix der Länge T_G [18].	10
2.5	Zeit-/Frequenzbereichsdarstellung eines OFDM-Symbols mit Cyclic Prefix. Das Cyclic Prefix schützt gegen Laufzeitverzögerungen und sichert die Orthogonalität der Subträger [18].	12
2.6	Blockdiagramm der digitalen OFDM-Senderkette	14
2.7	Ideale Konstellationsdiagramme von 16-QAM, 64-QAM und 256-QAM	16
2.8	Schematischer Frameaufbau in IEEE 802.11a/g [14]	18
2.9	Blockdiagramm der digitalen OFDM-Empfängerkette	20
2.10	Einfluss von AWGN auf die Konstellationsdiagramme der Modulationsarten 16-, 64- und 256-QAM.	21
2.11	Hohe Spitzenwerte im OFDM-Signal, die durch die Summation mehrerer Sinuswellen erzeugt werden [50]	24
2.12	Encoder des (2, 1, 4)-Faltungscodes in Schieberegister-Darstellung.	26
2.13	HLS-DATAFLOW-Prinzip ohne (links) und mit (rechts) Pipelining [5]	30
2.14	DSP-Board UniDAQ2.DSP-ADDA mit integriertem TMS320C6747	33
2.15	FPGA-Board C.Module.Unidaq2-FPGA basierend auf Artix-7 XC7A100T	34
2.16	Signalverbindung zwischen DSP und FPGA über den EMIFA-Bus	36
2.17	Signalstruktur des AXI4-Stream-Interfaces [13]	39
4.1	Blockdiagramm der MATLAB-Simulation (Tx–Rx-Kette)	48

4.2	BER gegen E_b/N_0 für 16/64/256-QAM: Theorie (AWGN) und OFDM-Simulation mit $N = 128$	50
4.3	Subträgerstruktur bei $N = 128$ mit Pilot- und Datenträgern (reelles OFDM, $f_s = 100$ kHz).	53
4.4	OFDM-Zeitsignal mit CP (gelb markiert), $N = 128$	54
4.5	PSD-Vergleich für $N_{\text{FFT}} = 64, 128, 256$ bei $f_s = 100$ kHz.	55
4.6	Konzeptionelle Partitionierungsvarianten des OFDM-Systems (Geteilte Verarbeitung)	58
4.7	Konzeptionelle Partitionierungsvarianten des OFDM-Systems (Teilweise Auslagerung)	58
4.8	EMIFA-Interface-Domänenkonzepte (synchron)	63
4.9	Pufferkonzept des DSP-FPGA-Systems	64
4.10	Datenpfad-Integrationskonzept FIFO $\leftrightarrow$ AXI4-Stream	67
4.11	Systemarchitektur mit getrennten Taktdomänen und Steuerpfaden	73
5.1	Signalverlauf während eines Schreib- Lesevorgangs über EMIFA (DSP-Core)	77
5.2	Signalverlauf während eines Schreib- und Lesevorgangs über EMIFA (ED-MA)	80
5.3	Ablaufdiagramm der ADC- und DAC-Interrupt-Service-Routinen	83
5.4	Ablaufdiagramm der OFDM-Verarbeitung zwischen DSP und FPGA	85
5.5	Takt- und Domänenarchitektur des EMIF-Interfaces	87
5.6	ASMD-Diagramm des FIFO-zu-AXI4-Stream-Moduls	91
5.7	Simulationswellenform der FIFO-zu-AXI4-Stream Übertragung	92
5.8	ASMD-Diagramm des AXI4-Stream-zu-FIFO-Moduls	93
5.9	Simulationswellenform des AXI4-Stream-zu-FIFO-Moduls	94
5.10	Interner Loopback zur Integration der FIFO-AXI-Module	95
5.11	Zuordnung der 16 möglichen 4-Bit-Nibbles zu den 16-QAM Konstellationspunkten (Realteil-Bits) und (Imaginärteil-Bits)	100
5.12	Encoder eines (2, 1, 6)-Faltungscodes in Schieberegister-Darstellung	112
5.13	Schematische Integration des FEC-Encoders und -Decoders in die OFDM-Verarbeitungskette	113
5.14	Systemaufbau zur Gesamtsystemverifikation (DSP/FPGA)	115
6.1	Zeitlicher Ablauf der Verarbeitung in der Pipelining-Struktur mit verdeutlichtem FPGA-Leerlaufzeitfenster	117

6.2	Oszilloskopmessung zur Latenzanalyse: Eingangssignal (CH1), durchgeschliffenes Signal (CH2) und verarbeitetes Signal (CH3)	118
6.3	Amplitudengang eines Sweep-Signals (10–50kHz) am DAC-Ausgang: Kanal 1 (Blau) ohne digitale Verarbeitung, Kanal 2 (Rot) nach Durchlauf der OFDM-Verarbeitungskette.	119
6.4	Vergleich der Rechenzeiten für verschiedene FFT-Längen auf DSP und FPGA (logarithmische Darstellung)	121
6.5	Abhängigkeit der Verarbeitungszeit auf dem FPGA von der Taktfrequenz	122
A.1	BER gegen E_b/N_0 für 16/64/256-QAM: Theorie (AWGN) und OFDM-Simulation mit $N_{\text{FFT}} \in \{64\}$	138
A.2	BER gegen E_b/N_0 für 16/64/256-QAM: Theorie (AWGN) und OFDM-Simulation mit $N_{\text{FFT}} \in \{256\}$	138
A.3	Subträgerlinien (reelles OFDM) bei $N_{\text{FFT}} = 64$ mit Daten- und Pilotträgern.	139
A.4	Subträgerlinien (reelles OFDM) bei $N_{\text{FFT}} = 256$ mit Daten- und Pilotträgern.	139
A.5	Signalverlauf während eines Schreibvorgangs über EMIFA (DSP-Core) . .	140
A.6	Signalverlauf während eines Lesevorgangs über EMIFA (DSP-Core)	140

Tabellenverzeichnis

2.1	Symbolübersicht für die Beschreibung von OFDM-Signalen	13
4.1	Subträgerbelegung je FFT-Länge (reelles OFDM).	50
4.2	16-Bit-Nutzdatenwörter pro OFDM-Symbol (nur Datenträger) für verschiedene N und QAM-Ordnungen.	52
4.3	Abgeleitete Kenngrößen bei $f_s = 100$ kHz und $CP = 1/8$	53
5.1	Verwendete Tools und Plattformen zur Systementwicklung	75
5.2	Vergleich der Zugriffszeiten bei 100 Datenwörtern über EMIFA mittels DSP-Core vs. EDMA (gemessen per Oszilloskop)	81
5.3	Vitis HLS – Ressourcenschätzung für FFT-Längen $N \in \{64, 128, 256\}$ bei 16-Bit-Twiddle-Faktoren (Sender)	107
5.4	Vitis HLS – Ressourcenschätzung für FFT-Längen $N \in \{64, 128, 256\}$ bei 16-Bit-Twiddle-Faktoren (Empfänger)	107
5.5	Vitis HLS – Latenzschätzung in Taktzyklen für die OFDM-Sende- und -Empfangspfade bei FFT-Längen $N \in \{64, 128, 256\}$ und 16-Bit-Twiddle-Faktoren	108
5.6	Systemparameter des verwendeten OFDM-Systems	114
6.1	Vergleich der Verarbeitungszeiten zwischen DSP- und FPGA-basiertem OFDM-System (Angaben in μ s)	120
6.2	FPGA-Ressourcenauslastung für verschiedene FFT-Längen	123
A.1	Ziele des Projekts (1 = hoch, 2 = mittel, 3 = niedrig)	135
A.2	Funktionale Anforderungen (1 = hoch, 2 = mittel, 3 = niedrig)	136
A.3	Nicht-funktionale Anforderungen (1 = hoch, 2 = mittel, 3 = niedrig) . . .	137

Listings

2.1	HLS-Schleifenoptimierung mit UNROLL und PIPELINE	31
5.1	Speicherabbildung und Zugriff auf FPGA über EMIFA	76
5.2	EDMA-Setup für 1D-Blocktransfer zwischen DSP und FPGA	79
5.3	Adresszuweisung für FIFO- und Steuerregister im EMIFA-Adressraum . .	96
5.4	Schnittstellendefinition der HLS-Module (Ausschnitt)	98
5.5	Subträgerindizes (Daten, Piloten, Nullträger) für OFDM-System mit $N =$ 128	102

1 Einleitung

1.1 Einführung und Motivation

Die Umsetzung moderner Signalverarbeitungsketten auf eingebetteten Plattformen erfordert Methoden, die eine gezielte Aufgabenteilung zwischen Hardware und Software ermöglichen. Besonders bei latenzsensitiven Anwendungen mit hoher Rechenlast – wie der kontinuierlichen Verarbeitung von Orthogonal Frequency Division Multiplexing (OFDM)-Signalen – stoßen rein softwarebasierte Systeme schnell an ihre Leistungsgrenzen. Das Hardware-Software-Codesign adressiert dieses Problem durch eine systematische Partitionierung und erlaubt die gezielte Auslagerung rechenintensiver Komponenten auf spezialisierte Hardwareeinheiten. High-Level-Synthese (HLS) erweitert diesen Ansatz, indem sie die Entwicklung und Integration von Hardwaremodulen auf Register-Transfer-Ebene deutlich vereinfacht und damit auch in frühen Entwurfsphasen zugänglich macht.

Orthogonal Frequency Division Multiplexing eignet sich aufgrund seiner modularen Struktur, hohen Datenrate und konfigurierbaren Verarbeitungskette besonders gut als Referenzanwendung für solche Codesign-Experimente. Gleichzeitig bildet das System praxisnahe Einsatzszenarien hinsichtlich Rechenaufwand, Timing, Schnittstellenintegration und Parametrierbarkeit ab. Die vorliegende Arbeit nutzt ein OFDM-System exemplarisch, um die Eignung und den praktischen Nutzen von HLS im Rahmen eines Hardware-Software-Codesigns auf einer heterogenen DSP-FPGA-Plattform zu untersuchen.

Der gewählte Ansatz zielt nicht nur auf die vollständige Umsetzung und Analyse des Systems ab, sondern legt auch die Grundlage für weiterführende Projekte und Anwendungen, insbesondere im Kontext der Lehre im Bereich eingebetteter Signalverarbeitung.

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist die modellbasierte Umsetzung eines OFDM-Systems im Rahmen eines Hardware-Software-Codesigns auf einer heterogenen DSP-FPGA-Plattform unter Einsatz von High-Level-Synthese. Im Zentrum steht die systematische Entwicklung, Integration und Verifikation aller Verarbeitungskomponenten mittels HLS – unter Berücksichtigung funktionaler, struktureller und performanzrelevanter Aspekte. Schwerpunkte bilden dabei die Entwicklung einer durchgängigen Signalverarbeitungskette, die Integration leistungsfähiger Kommunikationsschnittstellen zwischen den Subsystemen, sowie die Bewertung der HLS-basierten Architektur im Hinblick auf funktionale Korrektheit, Ressourceneffizienz und Erweiterbarkeit. Hierdurch soll ein modular aufgebautes und nachvollziehbar dokumentiertes System geschaffen werden, das nicht nur zur Demonstration der Machbarkeit dient, sondern zugleich eine flexible Grundlage für zukünftige Weiterentwicklungen in Forschung und Lehre bildet.

1.3 Gliederung der Arbeit

Die Arbeit folgt einer konsequenten methodischen Struktur: Sie führt von den theoretischen Grundlagen über die Entwicklung des Systems bis hin zur praktischen Umsetzung und abschließenden Evaluation. In Kapitel 2 werden zentrale Grundlagen eingeführt, insbesondere die Prinzipien des OFDM, die verwendeten Kommunikationsschnittstellen sowie die High-Level-Synthese. Darauf aufbauend werden im Kapitel 3 die funktionalen und nichtfunktionalen Anforderungen an das Zielsystem formuliert und leitet daraus zentrale Bewertungskriterien ab. Kapitel 4 widmet sich der Ausarbeitung der Systemarchitektur, wobei insbesondere die Entwurfsentscheidungen und ihre Auswirkungen auf Struktur und Kommunikationsverhalten im Vordergrund stehen.

Kapitel 5 behandelt die konkrete Umsetzung auf der Zielplattform. Es beschreibt die Entwicklung der HLS-basierten Verarbeitungskomponenten, die Integration der DSP-FPGA-Schnittstellen sowie den Aufbau der vollständigen Signalverarbeitungskette. Kapitel 6 dokumentiert die Ergebnisse der funktionalen Verifikation und Leistungsanalyse – insbesondere im Hinblick auf Korrektheit, Effizienz und Ressourcenauslastung. Kapitel 7 schließt die Arbeit mit einer zusammenfassenden Bewertung ab und gibt einen Ausblick auf mögliche Weiterentwicklungen.

2 Grundlagen

2.1 Orthogonal Frequency-Division Multiplexing (OFDM)

Orthogonal Frequency-Division Multiplexing (OFDM) ist ein Mehrträgerverfahren, bei dem die verfügbare Bandbreite in viele schmalbandige Subträger unterteilt wird [59]. Jeder Subträger überträgt dabei einen Teil der Gesamtdaten, wobei die Trägerfrequenzen so gewählt sind, dass sie trotz spektraler Überlappung orthogonal zueinander bleiben. Diese Eigenschaft ermöglicht eine hohe spektrale Effizienz und reduziert die Inter-Symbol-Interferenz (ISI) deutlich – ein zentraler Vorteil gegenüber klassischen Einträgerverfahren. OFDM gilt heute als Schlüsseltechnologie in zahlreichen Kommunikationsstandards, darunter Wireless Local Area Network (WLAN; IEEE 802.11a/g/n/ac) [14], Long Term Evolution (LTE) [56], Digital Video Broadcasting – Terrestrial (DVB-T/T2) [23] sowie Powerline-Kommunikation [40]. Besonders in drahtlosen und breitbandigen Übertragungsszenarien überzeugt das Verfahren durch seine Robustheit gegenüber frequenzselektiven Kanälen und Mehrwegeausbreitung.

Im Rahmen dieser Arbeit steht OFDM nicht als Untersuchungsgegenstand im Fokus, sondern wird als exemplarisches Kommunikationssystem zur Evaluation von Designmethoden auf einer DSP-FPGA-Plattform herangezogen. Ziel ist es, die Eignung von High-Level-Synthese im Kontext von Hardware-Software-Codesign anhand eines praxisrelevanten Systems zu analysieren. Daher beschränken sich die folgenden Ausführungen auf die für das Verständnis notwendigen technischen Grundlagen.

2.1.1 Mathematische Grundlagen

Das Verfahren OFDM basiert auf dem Prinzip der Mehrträgerübertragung: Anstatt einen Datenstrom seriell über eine einzelne Trägerfrequenz zu senden, werden die Daten auf viele einzelne Träger (sogenannte Subträger) aufgeteilt und parallel übertragen [45]. Damit dies ohne gegenseitige Störung funktioniert, sind die Subträger so gewählt, dass sie mathematisch orthogonal zueinander sind – also keine Energie gegenseitig überlagern. Die Erzeugung des OFDM-Signals erfolgt im sogenannten komplexen Basisband. Das bedeutet, dass die Signalverarbeitung auf digitalen, zeitdiskreten Signalen arbeitet, die reale und imaginäre Signalanteile enthalten. Dadurch kann das Verhalten des Gesamtsystems effizient in Software und auf digitaler Hardware abgebildet werden.

Zentral für OFDM ist die Anwendung der **inversen schnellen Fourier Transformation** (IFFT) zur Erzeugung des Sendesignals und der **schnellen Fourier Transformation** (FFT) zur Auswertung am Empfänger. Diese Transformationen ermöglichen es, viele Subträger gleichzeitig in einem Rechenschritt zu verarbeiten [59].

Im weiteren Verlauf werden die grundlegenden mathematischen Zusammenhänge erläutert, die dem OFDM-Signalmodell zugrunde liegen. Dabei werden unter anderem die Basisfunktionen, die verwendeten Transformationen und der Einfluss wichtiger Parameter wie Subträgeranzahl oder Symboldauer dargestellt.

Orthogonale Basisfunktionen

Ein zentrales Prinzip von OFDM ist die gleichzeitige Übertragung vieler Datensymbole auf separaten Frequenzkanälen, den sogenannten Subträgern. Damit diese sich nicht gegenseitig stören, nutzt OFDM spezielle mathematische Funktionen – orthogonale Basisfunktionen [38]. Jeder Subträger wird durch eine komplexe Exponentialfunktion beschrieben:

$$\phi_k[n] = e^{j\frac{2\pi kn}{N}}, \quad n = 0, 1, \dots, N-1 \quad (2.1)$$

Diese Funktionen sind paarweise orthogonal zueinander. Das bedeutet: Wenn man zwei verschiedene Subträger über ein ganzes Symbol betrachtet, ist ihr Überlapp (also ihr

inneres Produkt) gleich null:

$$\sum_{n=0}^{N-1} \phi_k[n] \cdot \phi_l^*[n] = \begin{cases} N, & \text{für } k = l \\ 0, & \text{für } k \neq l \end{cases} \quad (2.2)$$

Diese Eigenschaft ist die mathematische Grundlage dafür, dass Subträger trotz spektraler Überlappung keine gegenseitige Interferenz verursachen – solange Synchronisation in Zeit und Frequenz gewährleistet ist. Im Gegensatz zum klassischen Frequenzmultiplexing (FDM), bei dem zwischen den Trägern ein schützender Frequenzabstand eingehalten werden muss, nutzt OFDM die Orthogonalität zur spektralen Überlappung ohne gegenseitige Beeinflussung [53]. Dadurch kann die spektrale Effizienz deutlich gesteigert werden. Die gesamte verfügbare Bandbreite wird nahezu vollständig ausgenutzt, ohne dass sich Subträger gegenseitig stören – vorausgesetzt, die Orthogonalitätsbedingung wird exakt eingehalten, was eine präzise Synchronisation in Zeit und Frequenz erfordert.

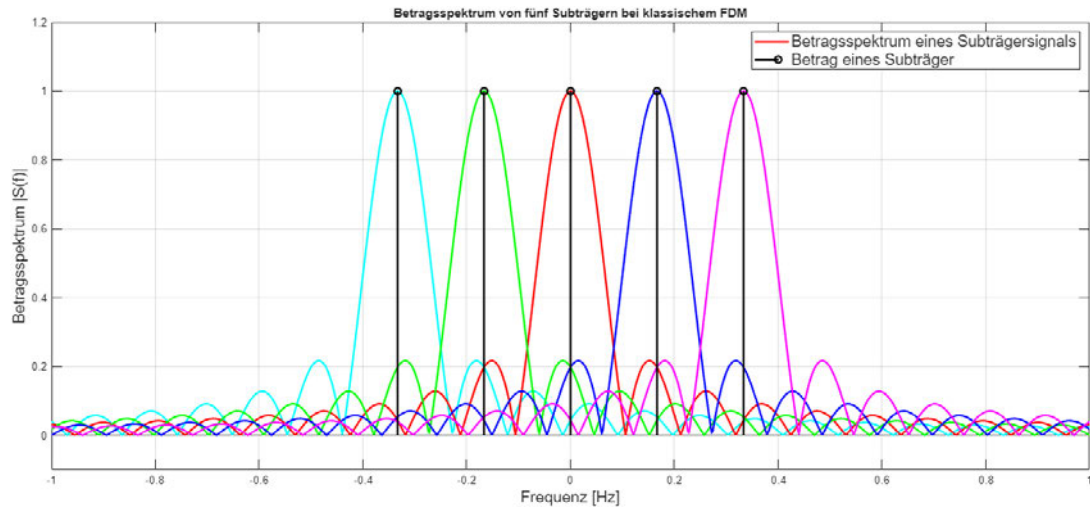


Abbildung 2.1: Betragsspektrum von fünf Subträgern bei klassischem FDM: Keine Orthogonalität, spektrales Übersprechen deutlich sichtbar.

Abbildung 2.1 zeigt exemplarisch das Spektrum klassischer FDM-Subträger, bei denen eine Frequenztrennung notwendig ist, um Interferenz zu vermeiden. Die Spektren überlappen, ohne orthogonale Nullstellenstruktur. Dies führt zu Interferenz zwischen benachbarten Subträgern.

Die Orthogonalität der Subträger funktioniert nur unter idealen Bedingungen [17]. In der Praxis kann es durch Zeit- oder Frequenzabweichungen zu Störungen kommen:

- **Inter-Carrier Interference (ICI):** ICI entsteht, wenn die Frequenzsynchronisation ungenau ist oder Dopplerverschiebungen im Kanal auftreten. In diesem Fall ist die Orthogonalität zwischen den Subträgern gestört, da sich ihre Spektren nicht mehr exakt an den Nullstellen der Nachbarn überlagern. Dies führt zu spektralem Übersprechen zwischen Subträgern, was eine fehlerfreie Demodulation erschwert.
- **Inter-Symbol Interference (ISI):** ISI tritt auf, wenn das Zeitsignal des Empfängers nicht exakt mit dem Beginn des OFDM-Symbols übereinstimmt, z. B. durch Laufzeitverzögerungen im Kanal oder unzureichende Synchronisation. Dadurch wird das empfangene Symbol durch das vorangegangene überlagert.

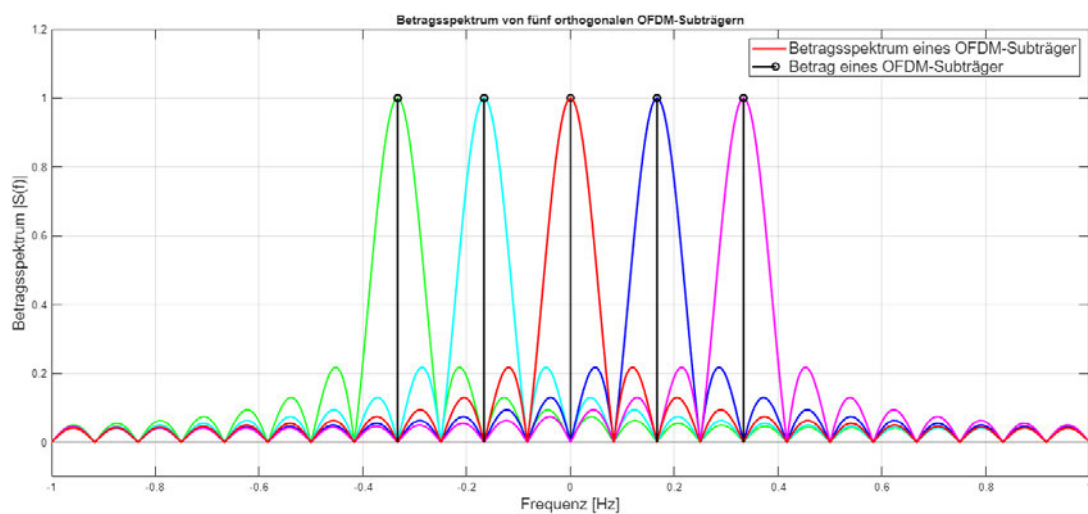


Abbildung 2.2: Betragsspektrum von fünf orthogonalen OFDM-Subträgern: Nullstellen der Nachbarn exakt getroffen – keine Interferenz trotz spektraler Überlappung.

Abbildung 2.2 zeigt die Spektren von orthogonalen OFDM-Subträgern. Obwohl sich die Spektren überlappen, tritt durch die mathematisch garantierte Nullstellenlage an den Frequenzpositionen der Nachbarn keine Interferenz auf. Diese Eigenschaft bildet die Grundlage für die hohe spektrale Effizienz von OFDM. Für die praktische Implementierung bedeutet das: Die Einhaltung der Orthogonalität gemäß Gleichung 2.2 hängt maßgeblich von einer präzisen Synchronisation in Zeit und Frequenz ab. Diese wird in der Regel durch spezielle Verfahren zur Taktsynchronisation und Trägerfrequenzkorrektur sowie durch den Einsatz von Referenzsymbolen realisiert.

Signalmodell eines OFDM-Symbols

Basierend auf den Basisfunktionen lässt sich ein OFDM-Symbol als eine gewichtete Summe mehrerer orthogonaler Subträger darstellen [59]. Jedes Modulationssymbol X_k belegt einen Subträger mit der Frequenz k/N . Das resultierende Zeitsignal $x[n]$ ergibt sich zu:

$$x[n] = \sum_{k=0}^{N-1} X_k \cdot e^{j\frac{2\pi kn}{N}}, \quad n = 0, 1, \dots, N-1 \quad (2.3)$$

Hierbei ist:

- N : Anzahl der Subträger,
- X_k : komplexes Datensymbol auf Subträger k ,
- n : Zeitindex innerhalb eines OFDM-Symbols.

Diese Darstellung entspricht der inversen diskreten Fourier Transformation (IDFT) der Modulationssymbole. In der Praxis wird zur effizienten Berechnung die IFFT eingesetzt. Das Signal $x[n]$ liegt im komplexen Basisband vor und wird vollständig digital erzeugt – ein wichtiger Vorteil für die Implementierung auf DSP- und FPGA-Plattformen.

OFDM-Signal als IDFT

Das zeitdiskrete OFDM-Signal $x[n]$ kann als Inverse Diskrete Fourier-Transformation (IDFT) der Symbolfolge $\{X_k\}$ aufgefasst werden:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{j\frac{2\pi kn}{N}}, \quad n = 0, \dots, N-1 \quad (2.4)$$

In der Praxis wird diese Transformation mithilfe der IFFT effizient berechnet. Die IFFT reduziert den Rechenaufwand von $\mathcal{O}(N^2)$ auf $\mathcal{O}(N \log(N))$ und ist damit besonders geeignet für hardwarenahe Implementierungen [24]. Aufgrund der Vielzahl an Multiplikationen und der streng sequenziellen Struktur zählt die IFFT zu den rechenintensivsten Komponenten einer OFDM-Verarbeitungskette. Für das Hardware-Software-Codesign ergeben sich daraus Anforderungen an Ressourcenbedarf, Latenzverhalten und Energieeffizienz – insbesondere bei der Realisierung auf FPGA oder DSP [36].

Hermitesymmetrie und reale Zeitsignale

Da das OFDM-Signal im komplexen Basisband erzeugt wird, enthält $x[n]$ in der Regel sowohl Real- als auch Imaginäranteile. Der Realteil beschreibt die In-Phase-Komponente (I), der Imaginärteil die Quadraturkomponente (Q). Diese Darstellung eignet sich hervorragend für digitale Verarbeitung und Simulation [59].

Für die physikalische Übertragung über reelle Kanäle – z. B. über DACs oder Lichtwellenleiter – ist jedoch ein reelles Zeitsignal erforderlich. Um ein solches reelles $x[n]$ zu erzeugen, müssen die Spektralkoeffizienten X_k hermitesymmetrisch sein:

$$X_k = X_{N-k}^*, \quad \text{für } 1 \leq k < N \quad (2.5)$$

Diese Bedingung stellt sicher, dass der Imaginäranteil im Zeitbereich vollständig verschwindet und $x[n] \in \mathbb{R}$ gilt. Die Konsequenz: Nur etwa die Hälfte der Subträger kann frei belegt werden. Die restlichen ergeben sich automatisch durch die Symmetrie. Diese Einschränkung muss bei der Dimensionierung von Datenraten und Systemparametern berücksichtigt werden, insbesondere bei der Auswahl der FFT-Größe und der Bandbreitennutzung.

Zeit- und Frequenzraster

Die Struktur eines OFDM-Signals basiert auf einem regelmäßigen Raster in Zeit und Frequenz. Dieses Raster ergibt sich aus zwei zentralen Systemparametern: der IFFT-Länge N (also der Anzahl an Subträgern) und der digitalen Abtastrate F_s . Daraus lassen sich die folgenden Größen ableiten:

$$\text{Subträgerabstand:} \quad \Delta f = \frac{F_s}{N} \quad (2.6)$$

$$\text{Dauer des OFDM-Nutzsymbols:} \quad T_u = \frac{1}{\Delta f} = \frac{N}{F_s} \quad (2.7)$$

$$\text{Dauer des Cyclic Prefix:} \quad T_{\text{CP}} = \frac{N_{\text{CP}}}{F_s} \quad (2.8)$$

$$\text{Gesamtdauer eines OFDM-Symbols:} \quad T_{\text{sym}} = T_u + T_{\text{CP}} \quad (2.9)$$

Diese Parameter bestimmen die Positionen der Symbole im Zeitbereich und der Subträger im Frequenzbereich. Sie sind nicht nur für die Signalbeschreibung essenziell, sondern auch für die Bewertung im Codesign-Kontext – etwa bei der Berechnung der erreichbaren Datenrate oder der spektralen Effizienz.

Einfluss der Subträgeranzahl N

Die Wahl der Subträgeranzahl N hat Einfluss auf die Eigenschaften eines OFDM-Systems [38]. Insbesondere folgende Abhängigkeiten sind relevant:

- **Trägerabstand:** Je größer N bei gegebener Abtastrate F_s , desto kleiner ist der Abstand Δf zwischen benachbarten Subträgern. Dies erhöht die spektrale Effizienz, macht das System jedoch empfindlicher gegenüber Frequenzabweichungen, etwa durch Dopplerverschiebung oder Trägerfrequenzfehler.
- **Symboldauer und ISI-Robustheit:** Ein größeres N verlängert die Nutzsymboldauer T_u . Dies verbessert die Robustheit gegenüber Mehrwegeausbreitung, da sich der Einfluss von Laufzeitunterschieden zwischen Pfaden relativ verringert. Gleichzeitig reduziert sich jedoch die zeitliche Auflösung.
- **Rechenaufwand:** Die IFFT-Operation zur Symbolerzeugung erfordert Rechenaufwand in der Größenordnung $\mathcal{O}(N \log(N))$. Ein höheres N bedeutet mehr Rechenlast, was sich auf die Latenz, Energieeffizienz und Ressourcennutzung bei der Implementierung auf FPGAs oder DSPs auswirkt [24].
- **PAPR-Verhalten:** Je mehr Subträger aktiv sind, desto größer ist die Wahrscheinlichkeit, dass sich mehrere Signalkomponenten konstruktiv überlagern. Dies führt zu hohen Spitzenauslenkungen im Zeitsignal – gemessen durch das *Peak-to-Average Power Ratio* (PAPR). Ein hoher PAPR belastet Leistungsverstärker, erfordert eine hohe Linearität und reduziert die Energieeffizienz [57].

Die Wahl von N ist daher ein kritischer Entwurfsparameter: Einerseits bestimmt sie die theoretische Leistungsfähigkeit des Systems, andererseits wirkt sie sich direkt auf die Komplexität und die Realisierbarkeit der Hardware aus. In der Praxis müssen Zielanforderungen wie Datenrate, Kanalbedingungen und Ressourcenbudget gemeinsam betrachtet werden, um eine geeignete Subträgeranzahl zu wählen.

Cyclic Prefix (CP) als Guard Interval

In realen Übertragungskanälen tritt häufig *Mehrwegeausbreitung* auf: wie in der Abbildung 2.3 zu erkennen ist, erreicht den Empfänger ein gesendetes Signal über mehrere Pfade mit unterschiedlichen Laufzeiten. Diese Laufzeitunterschiede führen dazu, dass sich Teile aufeinanderfolgender Symbole überlagern – ein Effekt, der als *ISI* bezeichnet wird. ISI verfälscht das empfangene Signal und erschwert eine saubere Trennung der OFDM-Symbole am Empfänger [18] [59].

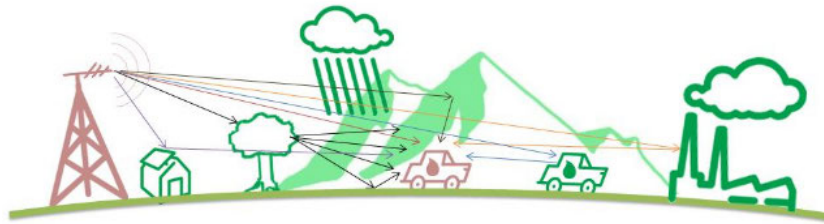


Abbildung 2.3: Einflüsse auf die Ausbreitung der elektromagnetischen Wellen im realen Mobilfunkkanal [31].

Um diesem Problem zu begegnen, wird bei OFDM ein sogenanntes *Guard Interval* eingeführt – meist in Form eines **Cyclic Prefix**. Dabei wird ein Teil des OFDM-Symbols kopiert und am Symbolanfang eingefügt, sodass das Gesamtsymbol aus zwei Abschnitten besteht: **Cyclic Prefix** mit N_{CP} Samples und eigentlichen OFDM-Nutzsymbol mit N Samples wie in der Abbildung 2.4 zu sehen ist, wobei mit T_G (Guard Interval) die Länge der T_{CP} und mit T_{sub} die Dauer des Nutzsymbols T_u gemeint ist.

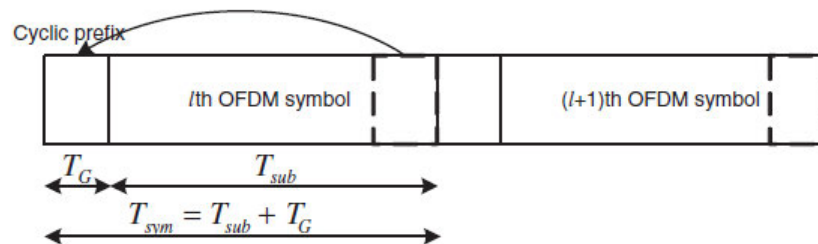


Abbildung 2.4: Zeitbereichsdarstellung zweier aufeinanderfolgender OFDM-Symbole mit Cyclic Prefix der Länge T_G [18].

Damit der Cyclic Prefix seine Schutzfunktion gegen ISI vollständig erfüllen kann, muss seine Dauer T_{CP} mindestens der maximalen Verzögerung des Kanals entsprechen. Ist dies erfüllt, liegt die Interferenz aus dem vorherigen Symbol vollständig im Cyclic Prefix-Teil

des aktuellen Symbols – der anschließend ohnehin entfernt wird. Die Auswertung mittels FFT über das reine Nutzsymbol bleibt dadurch unbeeinträchtigt.

Ein weiterer Vorteil: Durch die zyklische Wiederholung wird das lineare Faltungsergebnis des Übertragungskanals in eine *zyklische Faltung* überführt. Dadurch wird das Kanalverhalten im Frequenzbereich durch eine elementweise Multiplikation modelliert, was eine einfache Entzerrung ermöglicht [18].

Sei $x[n]$ das OFDM-Signal im Zeitbereich, $h[n]$ die Kanalimpulsantwort, und

$$y[n] = x[n] * h[n] + z[n] \quad (2.10)$$

das empfangene Signal, wobei $*$ die lineare Faltung und $z[n]$ additives Rauschen (z. B. AWGN) darstellen. Wird der Cyclic Prefix korrekt entfernt und eine FFT über den Nutzsymbolbereich angewendet, ergibt sich im Frequenzbereich:

$$Y_k = X_k \cdot H_k + Z_k \quad (2.11)$$

Hierbei sind:

- X_k : das gesendete Modulationssymbol auf Subträger k
- H_k : die komplexe Kanalfrequenzantwort auf Subträger k
- Z_k : die DFT des Rauschsignals $z[n]$

Die geschätzten Übertragungssymbole $\hat{X}_k$ lassen sich durch einfache komplexe Division rekonstruieren:

$$\hat{X}_k = \frac{Y_k}{\hat{H}_k} = \frac{X_k \cdot H_k + Z_k}{\hat{H}_k} \quad (2.12)$$

Dabei ist $\hat{H}_k$ eine im Empfänger geschätzte Kanalfrequenzantwort, z. B. durch Auswertung von Pilot-Subträgern. Diese sind spezielle, bekannte Symbole, die zur Schätzung von Kanalverzerrungen dienen.

Die Qualität der Schätzung $\hat{X}_k$ hängt maßgeblich von zwei Faktoren ab:

- der Genauigkeit der Kanalabschätzung $\hat{H}_k$
- dem Störabstand zwischen Signal- und Rauschanteil Z_k auf jedem Subträger

Subträgerabschaltung am Spektralrand:

Um unerwünschte Störungen angrenzender Kanäle zu vermeiden – etwa in Multi-User- oder Mehrkanal-Systemen – werden bei praktischen OFDM-Systemen die Subträger am unteren und oberen Spektralrand gezielt auf Null gesetzt. Diese *Spectral Guard Bands* schaffen Abstand zwischen aktiven Daten-Subträgern und benachbarten Frequenzbereichen. Damit wird spektrales Übersprechen reduziert, insbesondere bei begrenzter Linearität des Sendeverstärkers. In typischen Standards wie LTE oder WLAN sind diese Randbereiche standardisiert deaktiviert, z. B. 6 oder 12 Subträger an jedem Ende [18]. Abbildung 2.5 zeigt das Prinzip: Im Zeitbereich ist der Cyclic Prefix als Wiederholung des

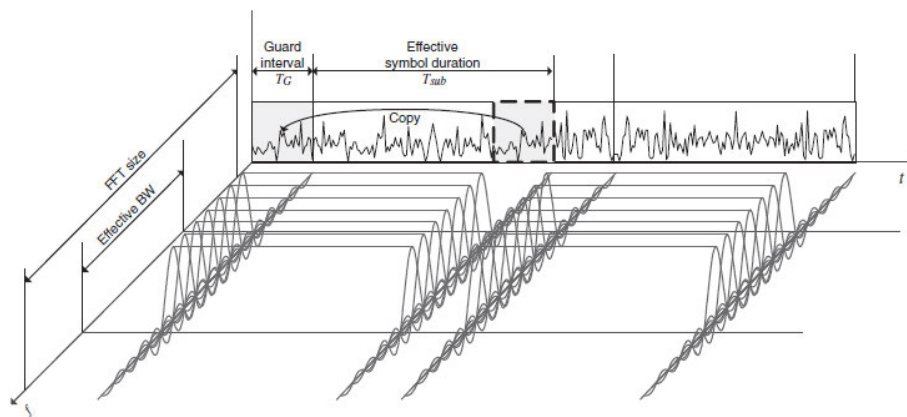


Abbildung 2.5: Zeit-/Frequenzbereichsdarstellung eines OFDM-Symbols mit Cyclic Prefix. Das Cyclic Prefix schützt gegen Laufzeitverzögerungen und sichert die Orthogonalität der Subträger [18].

Endes sichtbar. Im Frequenzbereich hat der Cyclic Prefix-Einsatz keinen direkten Einfluss auf das Spektrum des Symbols, da nur die Dauer verändert wird – nicht die IFFT selbst. Die symbolinterne Orthogonalität der Subträger bleibt erhalten, solange das Cyclic Prefix korrekt entfernt und die FFT exakt auf den Nutzsymbollbereich angewendet wird.

Darüber hinaus ist im Frequenzbereich auch erkennbar, dass die äußeren Subträger des IFFT-Eingangs explizit auf Null gesetzt wurden. Dies dient dazu, benachbarte Kanäle nicht zu stören und unerwünschte spektrale Nebenkeulen am Rand des genutzten Frequenzbereichs zu reduzieren. Diese Maßnahme verbessert die spektrale Verträglichkeit zu benachbarten OFDM-Systemen oder anderen Diensten und ist ein integraler Bestandteil der praktischen OFDM-Spektrumplanung.

Notation

Die folgende Tabelle 2.1 fasst die in diesem Abschnitt verwendeten Symbole zusammen, die für die Beschreibung und Analyse von OFDM-Systemen relevant sind. Sie dient als Referenz für die mathematischen Gleichungen und Konzepte im weiteren Verlauf.

Symbol	Bedeutung
N	Anzahl der Subträger / Länge der IFFT
N_{data}	Anzahl der Datensubträger
N_{Pilot}	Anzahl der Pilotensubträger
N_{CP}	Anzahl der Cyclic Prefix Subträger
X_k	Modulationssymbol auf Subträger k
$x[n]$	Zeitbereichssignal eines OFDM-Symbols
F_s	Digitale Abtastrate (Samples pro Sekunde)
Δf	Subträgerabstand
T_u	Dauer des Nutzsymbols (ohne Cyclic Prefix)
T_{CP}	Dauer des Cyclic Prefix
T_{sym}	Gesamtdauer eines OFDM-Symbols (inkl. Cyclic Prefix)

Tabelle 2.1: Symbolübersicht für die Beschreibung von OFDM-Signalen

2.1.2 OFDM-Sendekette und Frameaufbau

Übersicht der OFDM-Senderkette:

Die digitale Senderkette bildet die praktische Umsetzung des zuvor beschriebenen OFDM-Signalmodells [18]. Ziel ist es, aus einem bitseriellen Datenstrom ein übertragbares Zeitbereichssignal zu generieren, das den Anforderungen an spektrale Effizienz, Robustheit und Echtzeitverarbeitung genügt. Abbildung 2.6 zeigt eine vereinfachte Darstellung der zentralen Verarbeitungsschritte eines typischen OFDM-Senders.

Die Kette beginnt mit einer **Bitquelle**, die die zu übertragenden Nutzdaten bereitstellt. Optional kann eine **Quellcodierung** zur Reduktion der Datenrate oder Formatierung eingesetzt werden. Aufgrund der Implementierungskomplexität der Quellcodierung wird dieser Schritt in dieser Arbeit nicht weiter betrachtet. Im Anschluss folgt eine **Fehlerkorrekturcodierung (FEC)**. Typische Verfahren sind Faltungscodes, Turbo- oder

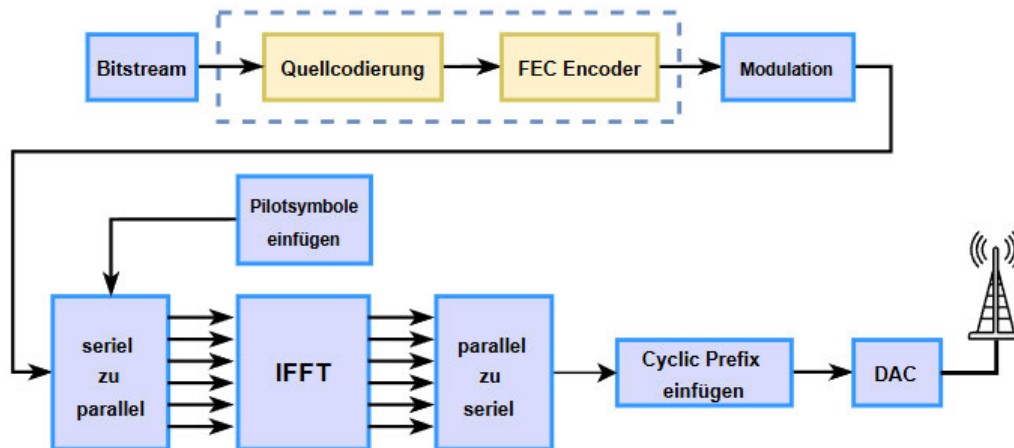


Abbildung 2.6: Blockdiagramm der digitalen OFDM-Senderkette

LDPC-Codes, abhängig vom Übertragungsstandard. Die FEC fügt gezielt Redundanz hinzu, um Übertragungsfehler im Empfänger erkennen und korrigieren zu können. Für diese Arbeit wird exemplarisch ein Faltungsdecoder implementiert, um die Integrationsfähigkeit im Rahmen eines Hardware-Software-Codesigns zu bewerten. In Abschnitt 2.2 werden Faltungscode und Decoder näher erläutert.

Nach der Codierung erfolgt die **Modulation**, bei der Bitgruppen in komplexwertige Symbole überführt werden. Die Auswahl der Modulationsart orientiert sich an den Anforderungen hinsichtlich Datenrate und Kanalqualität. Die modulierten Symbole werden in einem nächsten Schritt auf Subträger verteilt. Diese Subträgerbelegung folgt einem festen Raster, das auch **Pilot-Subträger** für Synchronisation und Kanalschätzung enthält. Wie bereits im Abschnitt zur Hermitesymmetrie dargestellt, wird nur ein Teil der Subträger aktiv genutzt. Die Randbereiche bleiben zur Spektrumsbegrenzung ungenutzt (vgl. Abschnitt 2.5). Dann erfolgt eine **Seriell-zu-Parallel-Umwandlung**, bei der N aufeinanderfolgende Samples in einen Vektor gebündelt werden.

Zur Transformation in den Zeitbereich wird eine IFFT durchgeführt. Die IFFT erzeugt das komplexe Basisbandsignal mit überlagerten, aber orthogonalen Subträgern – ein zentrales Merkmal von OFDM. Nach der **IFFT** erfolgt die **Parallel-zu-Seriell-Umwandlung**. Um ISI infolge von Mehrwegeausbreitung zu vermeiden, wird ein Cyclic Prefix angehängt. Dieser dient, wie zuvor in Abschnitt 2.1.1 erläutert, als Schutzintervall und ermöglicht gleichzeitig die effiziente Entzerrung im Frequenzbereich.

Das resultierende digitale Zeitsignal kann optional weiteren Verarbeitungsschritten wie Fensterung oder Upsampling unterzogen werden. Abschließend erfolgt eine **Digital-Analog-Wandlung (DAC)** und ggf. eine Hochfrequenzmodulation zur physikalischen Übertragung über den Zielkanal.

Digitale Modulation in OFDM-Systemen:

Im Rahmen der digitalen Modulation werden Bitgruppen aus dem Datenstrom in komplexe Symbole überführt, die über einen Träger gesendet werden [59]. In OFDM-Systemen erfolgt dies für jeden Subträger separat, wodurch sich eine hohe Parallelität und spektrale Effizienz erzielen lässt. Je nach Systemanforderung und Kanalqualität kommen dabei unterschiedliche Modulationsverfahren zum Einsatz – von einfachen Verfahren mit hoher Robustheit bis hin zu komplexeren Varianten mit hoher Bitdichte.

Typische Modulationsverfahren in OFDM-Standards sind:

- **Binary Phase Shift Keying (BPSK)** – 1 Bit pro Symbol (zwei Phasen), sehr robust, geringe Datenrate
- **Quadrature Phase-Shift Keying (QPSK)** – 2 Bits pro Symbol (vier Phasen), gute Robustheit, moderate Datenrate
- **Quadrature Amplitude Modulation (QAM)** – kombiniert Phasen- und Amplitudenmodulation für höhere Bitdichte

Mit steigender Modulationsordnung kann mehr Information pro Symbol übertragen werden, was die Datenrate erhöht – jedoch auf Kosten der Störanfälligkeit. Aufgrund dieses Kompromisses und der Fähigkeit zur flexiblen Anpassung an die Kanalbedingungen hat sich QAM als Standardlösung in modernen OFDM-Systemen etabliert.

Quadraturamplitudenmodulation (QAM):

Die Quadraturamplitudenmodulation (QAM) kombiniert Amplituden- und Phasenmodulation, um komplexwertige Symbole im zweidimensionalen I/Q-Raum zu codieren. Dabei wird jedes Symbol durch zwei orthogonale Komponenten beschrieben: die **In-Phase-Komponente (I)** und die **Quadratur-Komponente (Q)** [18].

In OFDM-Systemen wird QAM verwendet, um die Modulationssymbole auf die einzelnen Subträger abzubilden. Je nach Standard und Kanalqualität kommen unterschiedliche Modulationsordnungen zum Einsatz, z. B.

- **16-QAM:** 4 Bits pro Symbol, 16 Konstellationspunkte
- **32-QAM:** 5 Bits pro Symbol, 32 Konstellationspunkte
- **64-QAM:** 6 Bits pro Symbol, 64 Konstellationspunkte
- **256-QAM:** 8 Bits pro Symbol, 256 Konstellationspunkte

Abbildung 2.7 zeigt die Konstellationsdiagramme dieser drei Modulationsarten 16-, 64- und 256-QAM. Jedes Punktmuster bildet die möglichen Symbolpositionen im komplexen I/Q-Diagramm ab.

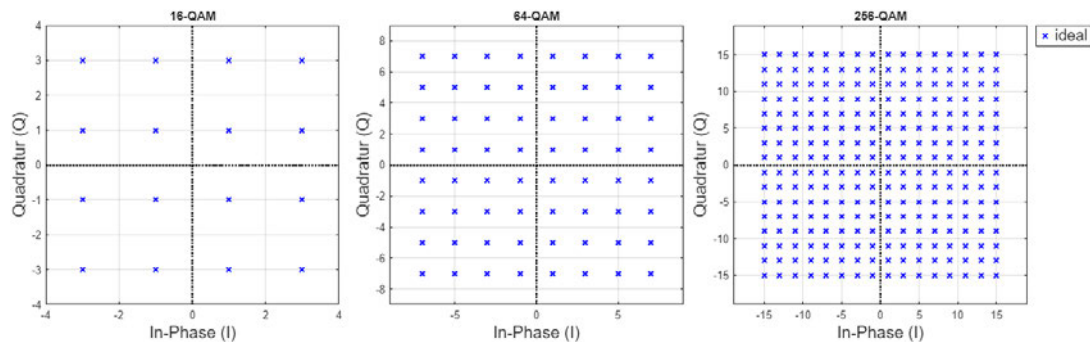


Abbildung 2.7: Ideale Konstellationsdiagramme von 16-QAM, 64-QAM und 256-QAM

Mit zunehmender Modulationsordnung steigt die Anzahl der Symbole im Konstellationsdiagramm exponentiell. Dadurch wächst die Datenrate pro Subträger, da mehr Bits pro Symbol übertragen werden. Gleichzeitig verringert sich der Abstand zwischen benachbarten Symbolen, was die Empfindlichkeit gegenüber Rauschen und Verzerrungen erhöht. Die Symbole liegen dichter im I/Q-Raum, wodurch sich bereits kleine Störungen deutlich auf die Fehlerwahrscheinlichkeit auswirken können [14].

In modernen OFDM-Standards wie LTE oder IEEE 802.11ac wird daher häufig eine adaptive Modulation eingesetzt: Die Modulationsstufe – also z. B. der Übergang von 16-QAM auf 64-QAM oder 256-QAM – wird dynamisch an die aktuelle Kanalqualität angepasst. Dies ermöglicht eine flexible Steuerung der Datenrate und erlaubt es dem System, bei guten Bedingungen hohe Durchsatz zu erreichen, ohne bei schlechten Bedingungen die Fehlerrate unnötig zu steigern.

Pilot-Subträger und Synchronisationssymbole:

In OFDM-Systemen kommen neben Nutzdaten auch bewusst platzierte Referenzsymbole zum Einsatz – darunter **Pilot-Subträger** und **Synchronisationssymbole**. Beide dienen der zuverlässigen Kanalrekonstruktion und der korrekten Symbolzuordnung am Empfänger.

Pilot-Subträger sind speziell reservierte Subträger, die mit bekannten Modulationsymbolen belegt werden [53]. Sie werden periodisch in Zeit und Frequenz verteilt, um dem Empfänger eine kontinuierliche Schätzung der Kanalimpulsantwort zu ermöglichen. Auf Basis dieser bekannten Referenzsymbole können Phasenverschiebungen, Amplitudendämpfungen und Frequenzgänge des Übertragungskanals zuverlässig geschätzt werden – auch bei dynamischen Kanalbedingungen wie Mobilität oder Mehrwegeausbreitung.

Die Anordnung und Dichte der Piloten ist standardabhängig. In IEEE 802.11a sind beispielsweise vier Subträger pro OFDM-Symbol als Piloten definiert [14]. Diese folgen einem festen Raster im sogenannten *Resource Grid*.

- **Hohe Pilotdichte:** Genauere Kanalschätzung, aber geringere spektrale Effizienz
- **Geringe Pilotdichte:** Höhere Datenrate, aber anfälliger bei schnellem Kanalwandel

Synchronisationssymbole hingegen dienen der Zeit- und Frequenzsynchronisation des Empfängers. Sie werden meist als spezielle OFDM-Symbole an definierten Stellen im Frame eingefügt – etwa zu Beginn eines Datenpakets. Diese Symbole weisen typische Korrelationseigenschaften auf (z. B. Schmidl-Cox oder Zadoff-Chu-Sequenzen), die eine robuste Erkennung und Justierung von Symboltiming, Trägerfrequenzfehlern oder Sampling-Offset ermöglichen [18]. Ohne zuverlässige Synchronisation wäre die Subträgerorthogonalität nicht gewährleistet – eine wesentliche Voraussetzung für die fehlerfreie Demodulation im OFDM-Empfänger [41].

Allgemeiner Aufbau eines OFDM-Frames:

Ein OFDM-Frame besteht typischerweise aus einer Abfolge definierter Symbolblöcke, die unterschiedliche Aufgaben im Übertragungsprozess übernehmen. Die grundlegende Struktur gliedert sich in drei funktionale Bereiche:

1. Preamble – dient der Synchronisation und Kanalschätzung am Empfänger
2. SIGNAL-Feld – enthält Steuerinformationen zur Decodierung des nachfolgenden Datenblocks
3. Nutzdatenbereich (Payload) – transportiert die eigentlichen Informationen

Diese Struktur ist in nahezu allen OFDM-basierten Standards zu finden, auch wenn Dauer, Modulationsart und Symbolanzahl je nach Standard unterschiedlich ausfallen.

Der folgende Abschnitt beschreibt den konkreten Frameaufbau am Beispiel von IEEE 802.11a/g – einem klassischen OFDM-Standard für WLAN-Systeme.

Frameaufbau in IEEE 802.11a/g-Systemen

Der Frameaufbau in OFDM-basierten IEEE 802.11a/g-Systemen folgt einem standardisierten Strukturmuster aus Vorlauf- und Nutzdatenbereich. Dieses dient der Initialisierung des Empfängers sowie der Übertragung der eigentlichen Nutzlastdaten. Wie in Abbildung 2.8 dargestellt, besteht der Frame aus drei Segmenten: den **Preamble**, **Signal-Feld** und dem **Payload-Feld**.

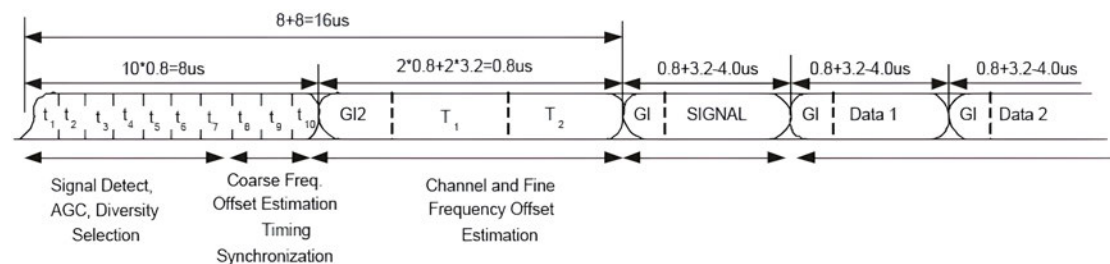


Abbildung 2.8: Schematischer Frameaufbau in IEEE 802.11a/g [14]

1. Preamble: Der Preamble besteht aus zwei Hauptkomponenten – dem **Short Training Field (STF)** und dem **Long Training Field (LTF)**. In realen Empfängern dienen diese Felder der Zeitsynchronisation, Frequenzoffset-Kompensation und der Initialisierung der Kanalschätzung [20]. Auf eine Implementierung der Preamble-Funktionen wird in dieser Arbeit bewusst verzichtet, da der Fokus auf der Senderkette liegt und der Aufwand für Synchronisation und Kanalschätzung den Rahmen sprengen würde [14].

- **STF (Short Training Field):** Besteht aus 10 kurzperiodischen OFDM-Symbolen mit spezieller Subträgerstruktur. Dient der groben Zeitsynchronisation und Detektion des Framebeginns.
- **LTF (Long Training Field):** Enthält zwei identische OFDM-Symbole mit langer Periodizität zur Trägerfrequenzkorrektur und Kanalinitialisierung. Die Subträger sind exakt bekannt und ermöglichen eine zuverlässige Kanalschätzung.

2. SIGNAL-Feld: Das SIGNAL-Feld folgt direkt auf den Preamble und überträgt Steuereinformationen über die Modulation, Codierungsrate und die Länge des Payloads [41]. Es wird stets mit BPSK (Coderate 1/2) moduliert und ist damit besonders robust. Auch dieser Block bleibt in dieser Arbeit unimplementiert, da keine vollständige PHY-Logik abgebildet wird.

3. Payload (Datenbereich): Der Datenbereich umfasst eine Sequenz von OFDM-Symbolen, die gemäß den im SIGNAL-Feld definierten Parametern kodiert, moduliert und auf Subträger verteilt werden [41]. Dieser Teil steht im Zentrum der implementierten OFDM-Senderkette und bildet die Grundlage für die Evaluierung des Hardware-Software-Codesigns.

2.1.3 OFDM-Empfängerkette

Übersicht der OFDM-Empfängerkette:

Die digitale Empfangskette stellt das funktionale Gegenstück zur OFDM-Senderkette dar [18]. Sie verarbeitet das über den Kanal übertragene, gestörte Zeitsignal und rekonstruiert daraus die ursprünglichen digitalen Nutzdaten. Abbildung 2.9 zeigt die zentralen Verarbeitungsschritte eines typischen digitalen OFDM-Empfängers.

Am Anfang der Kette steht die **Analog-Digital-Wandlung (ADC)**, welche das empfangene analoge Signal in ein digital abtastbares Zeitbereichssignal umwandelt. Es folgt die **Synchronisation**, die zwei wesentliche Aufgaben erfüllt:

- **Zeitsynchronisation:** Lokalisierung des Symbolbeginns mithilfe der korrelativen Eigenschaften des Preambles.
- **Frequenzsynchronisation:** Kompensation von Trägerfrequenzfehlern, um Inter-Carrier Interference (ICI) zu vermeiden.

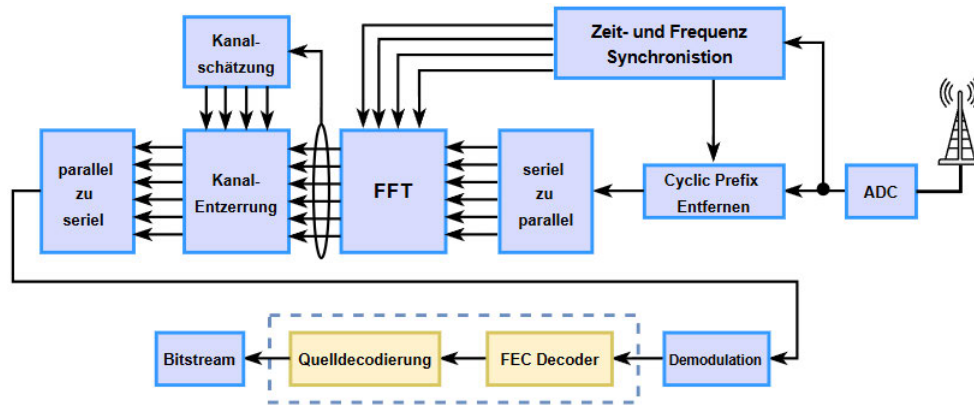


Abbildung 2.9: Blockdiagramm der digitalen OFDM-Empfängerkette

Diese Schritte basieren auf Trainingssequenzen (z. B. STF und LTF, siehe Abschnitt 2.8), werden in dieser Arbeit jedoch aus Komplexitätsgründen nicht implementiert.

Anschließend wird der **Cyclic Prefix** entfernt, sodass das verbleibende Signal exakt der IFFT-Länge entspricht. Dies stellt sicher, dass eine **FFT** über den reinen Nutzsymbolorbereich berechnet werden kann, was Voraussetzung für die Subträgerseparation und die darauffolgende Entzerrung ist.

Das resultierende Zeitsignal wird nun in den Frequenzbereich überführt. Dazu erfolgt zunächst eine **Seriell-zu-Parallel-Umwandlung**, bei der N aufeinanderfolgende Samples in einen Vektor gebündelt werden. Anschließend wird eine **FFT** durchgeführt, welche die parallelen Subträger Y_k extrahiert.

Die folgenden Verarbeitungsschritte dienen der Wiederherstellung der ursprünglichen Symbolfolge:

- **Kanalschätzung:** Auf Basis der bekannten Pilot-Subträger (siehe Abschnitt 2.1.2) wird die komplexe Kanalfrequenzantwort $\hat{H}_k$ bestimmt.
- **Entzerrung:** Jedes empfangene Symbol Y_k wird durch die geschätzte Kanalantwort geteilt:

$$\hat{X}_k = \frac{Y_k}{\hat{H}_k}$$

Dies hebt Verzerrungen durch Mehrwegeausbreitung und Frequenzgänge im Übertragungskanal auf.

Nach der **Entzerrung** erfolgt die **Parallel-zu-Seriell-Umwandlung**, sodass die Symbolfolge wieder bitweise weiterverarbeitet werden kann. In der **Demodulation** werden die komplexen QAM-Symbole in Bitgruppen zurückgeführt – analog zur Modulation in der Senderkette (siehe Abschnitt 2.1.2).

Darauf folgt die **Fehlerkorrekturdecodierung (FEC)**, in der verbliebene Bitfehler auf Basis der vorher eingeführten Redundanz korrigiert werden. Für diese Arbeit wird ein Faltungsdecoder verwendet. Details zur Architektur sind in Abschnitt 2.2 zu finden. Optional kann eine **Quell-Dekodierung** erfolgen, sofern am Sender eine Quellcodierung verwendet wurde. In dieser Arbeit wird dieser Schritt – wie in der Senderkette – nicht implementiert.

Am Ende der Kette steht der rekonstruierte **Bitdatenstrom**, der idealerweise mit dem ursprünglich gesendeten Datenstrom übereinstimmt. Die Güte der gesamten Empfangskette wird dabei durch die Kanalbedingungen, die Genauigkeit der Synchronisation sowie die Qualität der Kanalschätzung und Entzerrung bestimmt.

Einfluss von AWGN auf die QAM-Modulation

Die Robustheit digitaler Modulationsverfahren gegenüber Störungen hängt wesentlich vom Symbolabstand im Konstellationsdiagramm ab [25]. Mit steigender Modulationsordnung – z. B. bei 16-, 64- oder 256-QAM – erhöht sich die Anzahl der darstellbaren Symbole, während gleichzeitig die Abstände zwischen den einzelnen Symbolpunkten bei gleicher Gesamtleistung sinken. Dadurch steigt die Empfindlichkeit gegenüber Störungen wie Rauschen oder Kanalverzerrungen.

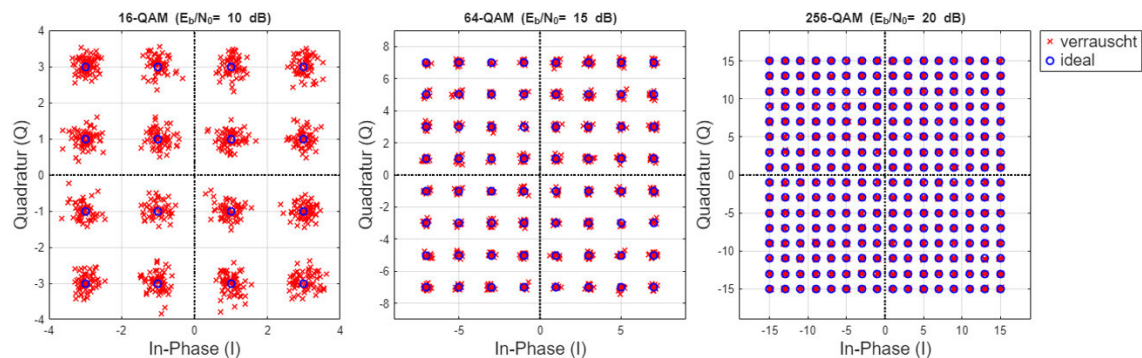


Abbildung 2.10: Einfluss von AWGN auf die Konstellationsdiagramme der Modulationsarten 16-, 64- und 256-QAM.

Abbildung 2.10 zeigt exemplarisch den Einfluss von **additivem weißen gaußschen Rauschen (AWGN)** auf drei QAM-Modulationsarten. In jedem Diagramm sind die empfangenen Symbolpunkte (**Rot**) als Streuung um die idealen Konstellationspositionen (**Blau**) dargestellt.

Um trotz zunehmender Modulationsdichte eine visuell deutliche Symboltrennung zu erhalten, wurde der *Energie-pro-Bit-zu-Rauschleistungsverhältnis* (E_b/N_0) jeweils angepasst:

- **16-QAM:** $E_b/N_0 = 10$ dB
- **64-QAM:** $E_b/N_0 = 15$ dB
- **256-QAM:** $E_b/N_0 = 20$ dB

Diese Konstellationsdiagramme verdeutlichen anschaulich, dass mit höherer Modulationsordnung eine deutlich höhere Signalgüte erforderlich ist, um die Symbole korrekt voneinander zu unterscheiden. Dies stellt besondere Anforderungen an die Kanalqualität, den Rauschabstand sowie die Genauigkeit der Synchronisation und Entzerrung [14].

Brutto- und Nettodatenrate in OFDM-Systemen

Die effektive Datenrate eines OFDM-Systems wird durch Systemparameter wie Subträgeranzahl, Modulationsordnung und symbolbezogenen Overhead bestimmt [14] [59]. Dabei unterscheidet man zwischen der **Bruttodatenrate** – also der maximal erreichbaren Datenrate unter idealen Bedingungen – und der **Nettodatenrate**, welche die tatsächlich nutzbare Datenmenge nach Abzug von Overhead und Redundanz beschreibt.

1. Bruttodatenrate:

Die Bruttodatenrate R_b gibt an, wie viele Bit pro Sekunde auf physikalischer Ebene übertragen werden können [18]. Sie lässt sich wie folgt berechnen:

$$R_b = N_{\text{data}} \cdot K \cdot \Delta f \cdot \frac{T_u}{T_{\text{sym}}} \quad (2.13)$$

mit:

- N_{data} : Anzahl der Subträger, die tatsächlich Nutzdaten tragen (exklusive Piloten, Schutzband und DC-Träger),

- K : Anzahl der Bits pro Symbol (z. B. 6 bei 64-QAM),
- Δf : Subträgerabstand (z. B. $\Delta f = 312.5 \text{ kHz}$ in IEEE 802.11a),
- T_u : Dauer des IFFT-Nutzsymbols (ohne Cyclic Prefix),
- $T_{\text{sym}} = T_u + T_{\text{CP}}$: Gesamtdauer eines OFDM-Symbols.

Der Term $\frac{T_u}{T_{\text{sym}}}$ berücksichtigt den Zeitanteil des Cyclic Prefix, der keine Nutzdaten trägt, aber zur Gesamtsymboldauer beiträgt. Je länger der Cyclic Prefix, umso größer ist der Overhead und desto geringer die effektive Bruttodatenrate. Die nutzbare Subträgeranzahl berechnet sich näherungsweise zu:

$$N_{\text{data}} = N - N_{\text{pilot}} - N_{\text{null}} - 1 \quad (2.14)$$

wobei:

- N : Gesamtanzahl der IFFT-Punkte (z. B. 64 bei IEEE 802.11a),
- N_{pilot} : Anzahl der Pilot-Subträger,
- N_{null} : Anzahl deaktivierter Subträger am Spektralrand (Schutzbänder),
- -1 : der zentrale DC-Träger, der typischerweise auf Null gesetzt wird.

2. Nettodatenrate: Die Nettodatenrate R_{netto} ergibt sich aus der Bruttodatenrate R_b unter Berücksichtigung zusätzlicher Protokoll- und Codierungsmechanismen:

$$R_{\text{netto}} = R_b \cdot R_{\text{FEC}} \quad (2.15)$$

mit:

- R_{FEC} : Coderate des Fehlerkorrekturverfahrens (z. B. $3/4$ bei einem FEC-Schema)

Die Nettodatenrate ist also stets kleiner als die Bruttodatenrate, reflektiert aber die real übertragbare Informationsmenge am Ausgang des Empfängers. In adaptiven Systemen wie WLAN oder LTE kann sie dynamisch an die Kanalbedingungen angepasst werden – z. B. durch Wahl einer niedrigeren Modulationsordnung oder einer robusteren Coderate bei schlechtem Empfang [14].

2.1.4 Peak-to-Average Power Ratio (PAPR)

Ein bekannter Nachteil von OFDM-Systemen ist das mitunter sehr hohe Verhältnis von Spitzen- zu Durchschnittsleistung – das sogenannte **Peak-to-Average Power Ratio (PAPR)** [57] [31]. Es resultiert aus der Überlagerung vieler Subträgersignale im Zeitbereich: Werden mehrere Subträger mit gleicher Phase konstruktiv summiert, entstehen Signalspitzen mit deutlich höherer Amplitude als der Mittelwert des Signals.

Das PAPR eines komplexen Zeitbereichssignals $x[n]$ mit N Samples ist definiert als:

$$\text{PAPR} = \max_{0 \leq n < N} \frac{|x[n]|^2}{\mathbb{E}[|x[n]|^2]} \quad (2.16)$$

Dabei bezeichnet $\max |x[n]|^2$ die maximale Momentanleistung des OFDM-Signals, während $\mathbb{E}[|x[n]|^2]$ den Erwartungswert (also die mittlere Leistung) über alle Samples beschreibt. Häufig wird der PAPR auch in Dezibel angegeben:

$$\text{PAPR}_{\text{dB}} = 10 \cdot \log_{10} \left(\frac{\max |x[n]|^2}{\mathbb{E}[|x[n]|^2]} \right) \quad (2.17)$$

Abbildung 2.11 veranschaulicht, wie die Amplitudenüberlagerung mehrerer Sinus-Unterträger kurzzeitig hohe Spitzen erzeugt und damit die PAPR erhöht [50].

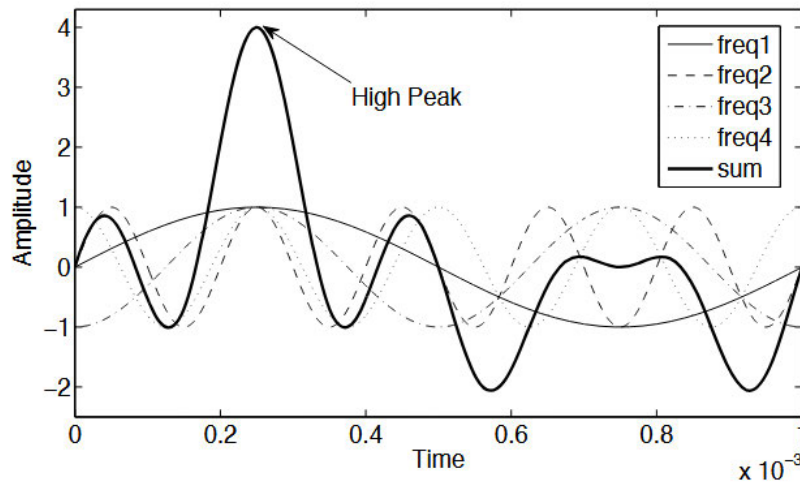


Abbildung 2.11: Hohe Spitzenwerte im OFDM-Signal, die durch die Summation mehrerer Sinuswellen erzeugt werden [50]

Ein hoher PAPR stellt insbesondere bei der analogen Leistungsverstärkung (z. B. im HF-Sender) ein Problem dar, da der Verstärker linear arbeiten muss, um Verzerrungen und Intermodulationen zu vermeiden. Dies reduziert die Energieeffizienz des Systems erheblich, da der Verstärker überdimensioniert oder mit großem Abstand zur Sättigungsgrenze betrieben werden muss [57].

Typische Maßnahmen zur PAPR-Reduktion:

- **Clipping:** Begrenzung der Amplituden auf einen Maximalwert – einfach, aber mit Verzerrungseffekten im Spektrum verbunden [29].
- **Selected Mapping (SLM) / PTS:** Mehrere OFDM-Signalvarianten mit identischem Inhalt werden erzeugt, und die Variante mit niedrigstem PAPR wird übertragen [29].
- **Codierungsbasierte Verfahren:** Symbolfolgen mit hohem PAPR werden bereits bei der Modulation vermieden.
- **Tone Reservation:** Einige Subträger werden gezielt so belegt, dass sie destruktiv zu PAPR-Spitzen wirken.

Obwohl keine dieser Maßnahmen ohne Nachteile ist (z. B. Rechenaufwand, Bandbreitenverlust oder zusätzliche Signalisierung), sind PAPR-Optimierungen in praktischen OFDM-Systemen sehr wichtig – insbesondere bei energiebegrenzten Anwendungen wie Mobilfunk oder WLAN-Endgeräten.

2.2 Faltungscodes

2.2.1 Grundidee und Einsatz

Faltungscodes sind eine Klasse von Fehlerkorrekturverfahren, die Datenströme nicht blockweise, sondern kontinuierlich codieren. Im Gegensatz zu Blockcodes hängt bei Faltungscodes das Ausgangssignal nicht nur vom aktuellen Eingabebit ab, sondern auch von einer festen Anzahl vorheriger Bits. Dies erzeugt Redundanz, die bei der Decodierung zur Erkennung und Korrektur von Bitfehlern genutzt wird [58].

Im Rahmen dieser Arbeit steht nicht die Implementierung eines produktiven Codiervorgangs im Vordergrund. Vielmehr soll anhand eines exemplarischen Faltungscodes die

grundsätzliche Funktionsweise solcher Codierer veranschaulicht und die Erweiterbarkeit des HLS-basierten Hardware-Software-Designs untersucht werden.

2.2.2 Encoderprinzip

Faltungscode-Encoder verarbeiten kontinuierliche Bitströme und erzeugen aus jedem Eingabebit eine feste Anzahl an Ausgabebits – unter Berücksichtigung vorheriger Eingaben. Die aktuelle Ausgangssequenz ist also eine gewichtete Funktion des aktuellen sowie der vorherigen $K - 1$ Bits, wobei K die Konstante Constraint Length bezeichnet. Der Encoder speichert diese Bits in einem Schieberegister und kombiniert sie über sogenannte Generatorpolynome.

Im Beispiel eines $(2, 1, 4)$ -Faltungscode werden aus jedem Eingabebit zwei Ausgabebits erzeugt – die Code-Rate beträgt damit $R = 1/2$. Der Index 4 steht für die Constraint Length, also die Tiefe des Schieberegisters (3 Speicher plus aktuelles Bit). Zur Erzeugung der Ausgabebits werden zwei Generatorpolynome verwendet:

$$G_1(X) = 1 + X^3 + X^4 \Rightarrow g_1[n] = \{1, 0, 0, 1, 1\} \quad (2.18)$$

$$G_2(X) = 1 + X + X^3 + X^4 \Rightarrow g_2[n] = \{1, 1, 0, 1, 1\} \quad (2.19)$$

Diese Polynome legen fest, welche Bits im Register jeweils über ein XOR verknüpft werden, um ein Ausgabebit zu erzeugen. Das erste Ausgabebit wird durch Anwendung von g_1 , das zweite durch g_2 gebildet.

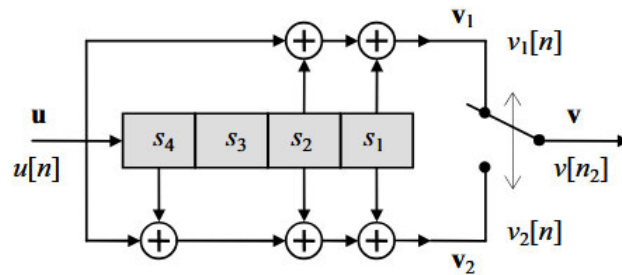


Abbildung 2.12: Encoder des $(2, 1, 4)$ -Faltungscode in Schieberegister-Darstellung.

Abbildung 2.12 zeigt die Implementierung des Encoders: Die eingehenden Bits werden schrittweise durch das Register geschoben. Bei jedem Takt erzeugt die Kombination der

aktuellen und zurückliegenden Bits – entsprechend den Generatorpolynomen – zwei codierte Ausgabebits. Dadurch wird der Informationsstrom redundant, aber strukturiert erweitert. Diese Struktur ist Voraussetzung für eine spätere Korrektur beim Empfänger. Der Encoder arbeitet deterministisch und ohne Feedback: Für jedes Eingabebit entsteht eine feste, vorhersagbare Bitfolge. Diese Eigenschaft ist besonders für hardwarefreundliche Implementierungen relevant, da der logische Aufbau einfach und in Echtzeit realisierbar ist.

2.2.3 Decodierung

Die Decodierung eines Faltungscodes erfolgt typischerweise mit dem **Viterbi Algorithmus** – einem effizienten Verfahren zur Maximum-Likelihood-Sequenzschätzung [58]. Ziel ist es, aus einer fehlerbehafteten Bitfolge die wahrscheinlichste ursprüngliche Informationssequenz zu rekonstruieren. Der Viterbi-Decoder analysiert dabei alle möglichen Zustandsfolgen im Trellis-Diagramm des Faltungscodes und berechnet für jede einen sogenannten Pfadmetrikwert. Dieser Wert gibt an, wie gut der jeweilige Pfad zur empfangenen Sequenz passt. Es wird stets nur der wahrscheinlichste Pfad (der mit der geringsten Metrik) bis zu einem bestimmten Zeitpunkt weiterverfolgt – was den Rechenaufwand deutlich reduziert.

Hard-Decision vs. Soft-Decision:

- **Hard-Decision:** Die empfangenen Bits werden vor der Decodierung in binäre Werte (0 oder 1) quantisiert. Die Entscheidung basiert also nur auf der Bitfolge, ohne Berücksichtigung der Signalqualität. Diese Variante ist einfach zu implementieren, aber weniger leistungsfähig.
- **Soft-Decision:** Die Decoder-Logik verarbeitet zusätzlich die Amplitudeninformationen der empfangenen Symbole. Dadurch kann der Decoder in kleinen Schritten die Wahrscheinlichkeiten bewerten und Fehlentscheidungen besser vermeiden – was zu einer besseren Fehlerkorrekturleistung führt, allerdings auf Kosten höherer Rechenkomplexität.

Die in diesem Abschnitt dargestellten Inhalte bieten eine grundlegende Einführung in die Funktionsweise von Faltungscodes. Ziel war es, das Prinzip hinter Codierung und Decodierung nachvollziehbar zu vermitteln und deren Rolle im Rahmen von Forward Error Correction (FEC) einzuordnen.

Die Betrachtung der praktischen Umsetzung – insbesondere im Hinblick auf die Implementierung eines exemplarischen Faltungscoders und -decoders mittels High-Level-Synthese – erfolgt im Kapitel 5.8.

2.3 High-Level-Synthese (HLS)

Die **High-Level-Synthese (HLS)** beschreibt den automatisierten Entwurfsprozess, bei dem eine algorithmische Beschreibung – in C oder C++ – in eine synthetisierbare Register-Transfer-Level (RTL)-Architektur überführt wird [16] [5]. Im Gegensatz zur klassischen HDL-basierten Entwicklung mit VHDL oder Verilog steht bei HLS die Funktionalität im Vordergrund, nicht die manuelle Strukturierung der Schaltung. Ziel ist es, den Hardware-entwicklungsprozess auf eine höhere Abstraktionsebene zu heben, um Entwicklungsaufwand, Komplexität und Fehleranfälligkeit deutlich zu reduzieren.

HLS erlaubt insbesondere bei komplexen signalverarbeitenden Algorithmen oder schnell ändernden Anforderungen eine effizientere Design-Iteration. Änderungen an der Funktionalität lassen sich direkt im Quellcode durchführen, ohne tiefgreifende manuelle RTL-Anpassungen vornehmen zu müssen. Zudem ermöglicht HLS eine gezielte Optimierung von Latenz, Durchsatz, Ressourcenverbrauch und Taktfrequenz durch **Pragmas** und automatisierte Analysen – und bietet dabei weiterhin ausreichend Einflussmöglichkeiten auf das erzeugte Hardwareverhalten [22].

2.3.1 Grundlagen der High-Level-Synthese

Der Kern der High-Level-Synthese besteht darin, eine Beschreibung auf C- oder C++ Ebene in ein äquivalentes Hardwaremodell umzusetzen [5]. Dabei werden typische Programmierkonstrukte wie Schleifen, Bedingungen, Funktionsaufrufe oder Datenstrukturen analysiert und in steuerbare Hardwarekomponenten überführt – wie Register, Multiplexer, Zustandsautomaten oder Recheneinheiten.

Typische Optimierungsziele im HLS-Prozess sind:

- **Minimierung der Latenz** – Reduktion der Zyklusanzahl pro Operation
- **Maximierung des Durchsatzes** – Parallelisierung durch Pipelining oder Task-Level-Parallelität

- **Effiziente Ressourcennutzung** – Einsparung von Logik, Speicher oder DSP-Blöcken
- **Taktfrequenzanpassung** – Strukturierung des Designs zur Einhaltung gegebener Timing-Ziele

HLS eignet sich besonders für datenflussorientierte Anwendungen wie Signalverarbeitung, Video-/Bildverarbeitung, kryptografische Funktionen oder Steuerungssysteme, bei denen klar strukturierte Algorithmen in feste Ablaufmuster gegossen werden können.

2.3.2 Werkzeuge und Design-Flow

Ein typischer HLS-Designflow beginnt mit einer funktionalen Beschreibung des Algorithmus in einer Hochsprache. Diese Beschreibung wird anschließend mit Hilfe von **Pragmas** gezielt für die Hardwareoptimierung vorbereitet – etwa durch Schleifen-Unrolling, Funktions-Inlining, Ressourcen-Sharing oder die Einführung von Pipeline Stufen. Ziel dieser **Pragmas** ist es, die Struktur des später erzeugten Hardwaremoduls hinsichtlich Latenz, Durchsatz und Ressourceneinsatz gezielt zu beeinflussen [61] [62] [8].

Nach der High-Level-Synthese erzeugt das HLS-Werkzeug automatisch eine RTL Beschreibung in VHDL. Diese kann direkt in klassische FPGA-Toolchains wie Vivado eingebunden werden – für die anschließende Implementation, Platzierung, Routing sowie die Bitstream-Generierung.

Ein besonderer Vorteil von HLS ist die Möglichkeit, bereits in der Hochsprache C/C++ vollständige **Testbenches** zu formulieren. Funktionale Tests und Simulationen können somit ohne Wechsel in eine HDL-Umgebung durchgeführt werden. Dies beschleunigt die Entwicklung und erlaubt eine frühzeitige Validierung der Algorithmusfunktionalität vor der eigentlichen Hardwareintegration.

Abbildung 2.13 veranschaulicht exemplarisch das Prinzip des **Dataflow-Modells** in HLS – einmal ohne (links) und einmal mit (rechts) Pipelining. Durch gezieltes Pipelining können mehrere Verarbeitungsschritte überlappend durchgeführt werden. Dies führt insbesondere in datenstrombasierten Anwendungen zu einem deutlichen Durchsatzgewinn, ohne die Rechenlogik mehrfach duplizieren zu müssen.

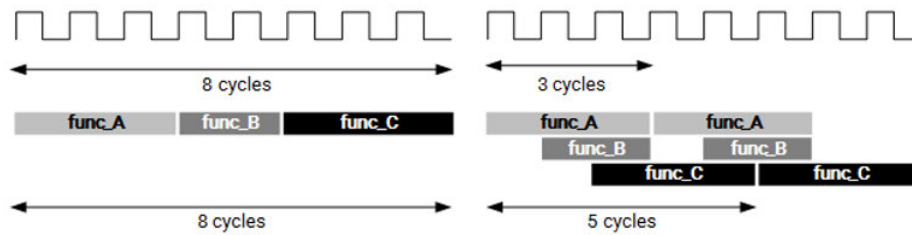


Abbildung 2.13: HLS-DATAFLOW-Prinzip ohne (links) und mit (rechts) Pipelining [5]

2.3.3 Pragmas und Optimierungsdirektiven in HLS

Damit HLS-Tools effektiven RTL-Code generieren, reichen rein funktionale Beschreibungen nicht aus. Über sogenannte **Pragmas** wird das Syntheseverhalten gezielt gesteuert. Diese Anweisungen geben dem Synthesewerkzeug konkrete Hinweise zur Strukturierung, Parallelisierung oder Ressourcenoptimierung – ohne dabei die Funktionalität des Quellcodes zu verändern [5] [22].

Nachfolgend sind zentrale **Pragmas** erläutert, die im HLS-Prozess regelmäßig zum Einsatz kommen:

- **#pragma HLS PIPELINE:**
Fügt eine Pipeline-Struktur in Schleifen oder Funktionsblöcke ein. Ziel ist ein hoher Datendurchsatz bei minimaler Initiationszeit. Dies ermöglicht eine überlappende Abarbeitung von Operationen.
- **#pragma HLS UNROLL:**
Entrollt Schleifen ganz oder teilweise. Dadurch werden mehrere Iterationen gleichzeitig ausgeführt, was parallele Hardwareeinheiten erzeugt – auf Kosten zusätzlicher Ressourcen.
- **#pragma HLS DATAFLOW:**
Aktiviert die automatische Parallelisierung mehrerer Funktionen oder Schleifenabschnitte, sofern zwischen ihnen FIFO-ähnliche Datenströme bestehen. Besonders nützlich bei datenstromorientierten Designs wie Signalverarbeitung oder Bildanalyse.
- **#pragma HLS INLINE:**
Zwingt das Inlining einer Funktion – also das Einfügen des Funktionskörpers an-

stelle eines Funktionsaufrufs. Dies kann helfen, kritische Pfade zu verkürzen und Pipelining über Funktionsgrenzen hinweg zu ermöglichen.

- **#pragma HLS ALLOCATION:**

Steuert die maximale Anzahl an Ressourcen (z. B. Addierer, Multiplizierer), die für bestimmte Operationen verwendet werden dürfen. Was eine explizite Kontrolle über Ressourcenteilung oder -begrenzung ermöglicht.

- **#pragma HLS RESOURCE:**

Erzwingt die Nutzung bestimmter FPGA-interner Ressourcen (z. B. DSP-Blöcke, Block-RAM) für eine bestimmte Operation oder Datenstruktur.

Beispiel: Der folgende Codeausschnitt 2.1 zeigt zwei Varianten einer Schleifenoptimierung mit HLS-Pragmas:

```
1 void add_loop(int A[8], int B[8], int C[8]) {
2
3     /* Variante A: Vollständiges Unrolling */
4     for (int i = 0; i < 8; i++) {
5         /* vollstaendige Parallelisierung, hoher Ressourcenverbrauch */
6         #pragma HLS UNROLL
7         C[i] = A[i] + B[i];
8     }
9
10    /* Variante B: Pipelining mit Initiation Interval = 1 */
11    for (int i = 0; i < 8; ++i) {
12        /* pipelinede Verarbeitung mit Durchsatz = 1 Ergebnis/Zyklus */
13        #pragma HLS PIPELINE II=1
14        C[i] = A[i] + B[i];
15    }
16 }
```

Listing 2.1: HLS-Schleifenoptimierung mit UNROLL und PIPELINE

In **Variante A** bewirkt das vollständige **#pragma HLS UNROLL**, dass alle 8 Additionseinheiten gleichzeitig realisiert werden. Dadurch kann das Ergebnis für alle 8 Werte in einem einzigen Taktzyklus berechnet werden. Diese maximale Parallelität geht jedoch mit einem hohen Ressourcenverbrauch einher, da jede Addition eine eigene Recheneinheit benötigt.

In **Variante B** hingegen wird mittels **#pragma HLS PIPELINE mit II=1** nur eine einzelne Additionseinheit verwendet. Diese wird durch Pipelining so gesteuert, dass in jedem

Takt ein neues Ergebnis berechnet wird. Dadurch reduziert sich der Ressourcenverbrauch deutlich – bei nur geringfügigem Durchsatzverlust für kleinere Schleifenlängen.

Dieses Beispiel zeigt exemplarisch den Trade-off zwischen Parallelität und Ressourceneffizienz, der bei der HLS-Optimierung durch geeignete Direktiven gezielt gesteuert werden kann.

Ohne gezielte Direktiven erzeugt das HLS-Tool standardmäßig eine ressourcensparende, aber oft leistungsschwache Hardwarearchitektur [5]. Erst durch gezielte Anwendung von Direktiven entsteht ein effizientes Hardwaredesign, das Timing-Anforderungen erfüllt und gleichzeitig Ressourcen sinnvoll nutzt. Die Auswahl und Kombination dieser Optimierungen ist dabei stark anwendungsabhängig. Eine gute Kenntnis dieser Werkzeuge ist wichtig für jeden, der HLS produktiv einsetzen möchte. Die im Rahmen dieser Arbeit gesammelten Erfahrungen mit dem Einsatz von High-Level-Synthese – insbesondere hinsichtlich der Handhabung von Optimierungspragmas, Ressourcensteuerung und funktionaler Validierung – werden im Kapitel 5.5 detailliert beschrieben.

2.4 Hardwareplattform

Für die Umsetzung und Evaluierung der OFDM-Architektur kommen in dieser Arbeit zwei aufeinander abgestimmte Hardwarekomponenten zum Einsatz: ein DSP-Board zur Datenerfassungs- und Signalverarbeitung sowie ein FPGA-Board für die hardwarebeschleunigte Signalverarbeitung. Beide Boards stammen vom selben Designer und sind über eine direkte Busverbindung gekoppelt [1].

2.4.1 DSP-Board TMS320C6747

Das eingesetzte DSP-Board **UniDAQ2.DSP-ADDA** basiert auf einem **TMS320C6747**, einem 32-/64-bit Floating-Point-DSP mit einer maximalen Taktfrequenz von 456 MHz. Das Board ist speziell für Anwendungen in der Mess-, Steuerungs- und Signalverarbeitung konzipiert und verfügt über integrierte AD- und DA-Wandler, Speicher sowie standardisierte Schnittstellen [1].

- **DSP:** TMS320C6747 (456 MHz)
- **Speicher:** 64 MB SDRAM

- **A/D-Wandler:** 16 gleichzeitig abtastende SAR-Wandler (16 Bit, 175 kS/s)
- **D/A-Wandler:** 8 monotone DA-Wandler (16 Bit, 105 kS/s)
- **Schnittstellen:** USB 2.0 Device, 100Base-T Ethernet
- **Erweiterbarkeit:** GPIOs, FPGA-Interface, Display, Massenspeicher

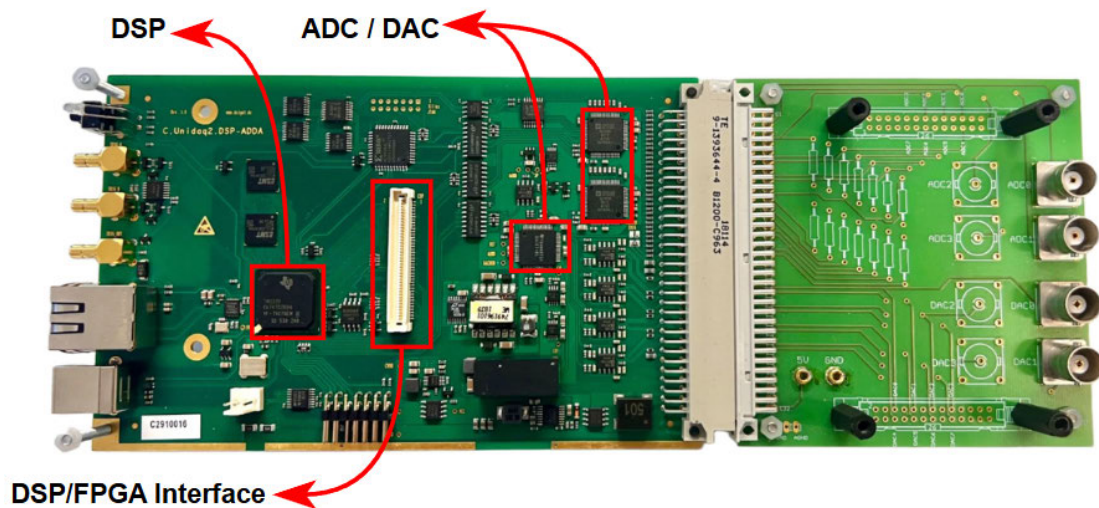


Abbildung 2.14: DSP-Board UniDAQ2.DSP-ADDA mit integriertem TMS320C6747

2.4.2 FPGA-Board: Artix-7 XC7A100T

Für die hardwareseitige Implementierung der HLS-basierten Signalverarbeitungsblöcke wird ein kompaktes **FPGA-Board** verwendet. Es basiert auf einem **Artix-7 XC7A100T** und ist über ein Interface eng mit dem DSP-Board gekoppelt [1].

- **FPGA:** Artix-7 XC7A100T
- **Taktung:** Externer 50 MHz-Oszillator (auch externe Clock möglich)
- **Interne Taktfrequenz:** Designabhängig, bis über 300 MHz
- **Kommunikation:** 16-Bit asynchrones Speicher-Interface zum DSP
- **Konfiguration:** Host SPI über Flash (auch DSP-gesteuert)

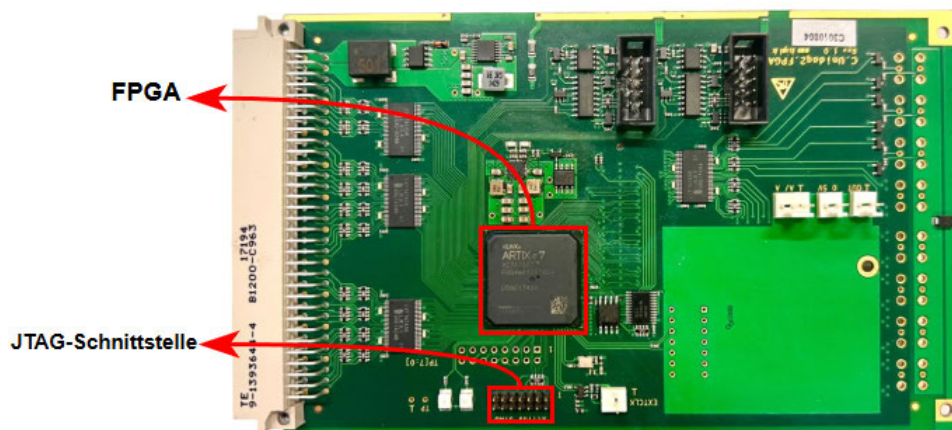


Abbildung 2.15: FPGA-Board C.Module.Unidaq2-FPGA basierend auf Artix-7 XC7A100T

Die Verbindung zwischen DSP und FPGA erfolgt über ein 16-Bit breites, asynchrones Speicher-Interface. Der DSP übernimmt hierbei die Rolle des Busmasters und adressiert den FPGA als speicherabbildbares Peripheriegerät. Der adressierbare Bereich umfasst 14 Bit, entsprechend 16.384 16-Bit-Worte.

Die Konfiguration des FPGA erfolgt im Host-SPI-Modus über eine dedizierte Flash. Alternativ kann der DSP über einen separaten SPI-Zugriff die FPGA-Konfiguration initialisieren, aktualisieren oder überwachen. Für beschleunigte Entwicklungszyklen, Tests oder Debugging kann der FPGA direkt über die **JTAG-Schnittstelle** programmiert werden. Dies ermöglicht eine unkomplizierte und schnelle Wiederkonfiguration ohne Umweg über den DSP oder Flash.

2.5 Asynchrones Bussystem EMIFA

Für die Kommunikation zwischen dem DSP und dem FPGA kommt in dieser Arbeit das External Memory Interface A **EMIFA**-Interface zum Einsatz – ein asynchrones Speicherbus-System, das vom DSP bereitgestellt und vom FPGA als Client adressiert wird [3] [4]. Der EMIFA-Bus ist für eine flexible Anbindung externer Speicher- und Peripheriegeräte konzipiert und bietet sowohl Lese- als auch Schreiboperationen mit konfigurierbarem Timing. In der hier verwendeten Architektur stellt er die zentrale Datenschnittstelle zwischen software- und hardwarebasierter Signalverarbeitung dar.

2.5.1 Signale, Timing und Betriebsarten

Der EMIFA-Bus des TMS320C6747-DSP arbeitet asynchron mit einem 16 Bit breiten Datenbus `DSP_DATA` und einem 14 Bit breiten Adressbus `DSP_ADDR`. Die Steuerung der Buszugriffe erfolgt über klassische Steuersignale wie:

- `DSP_CS2N` – Chip Select: aktiviert den FPGA-Bereich
- `DSP_WEN` – Write Enable: signalisiert einen Schreibvorgang
- `DSP_OEN` – Output Enable: signalisiert einen Lesevorgang
- `DSP_WAIT` – Wait Input: erlaubt es dem FPGA, Zugriffe zu verzögern
- `DSP_EMIF_CLK` – liefert den EMIFA-Systemtakt

Je nach Richtung des Datenflusses initiiert der DSP-Core entweder einen Schreib- oder Lesevorgang auf dem Bus. Dabei sind insbesondere die folgenden Zeitparameter entscheidend für die Busleistung:

- **Setup-Zeit** (t_{setup}): Zeitraum vor Aktivierung des Steuer-Signals, in dem Adress- und Datenleitungen stabil sein müssen.
- **Strobe-Zeit** (t_{strobe}): Dauer, während der die Datenübertragung durch aktive Steuerleitungen (z. B. `WEN/OEN`) stattfindet.
- **Hold-Zeit** (t_{hold}): Zeitraum nach dem Deaktivieren der Steuerleitung, in dem Signale gültig bleiben müssen.
- **Turn-Around-Zeit** (t_{TA}): Zeitraum zwischen zwei entgegengesetzten Busoperationen (z. B. Lesen $\rightarrow$ Schreiben oder Schreiben $\rightarrow$ Lesen), in dem der Bus in einem neutralen Zustand bleibt, damit keine Signalkollisionen auftreten und alle Leitungen sicher entkoppelt sind.

Das Handshake-Signal `WAIT` spielt auch eine Rolle: Der FPGA kann dieses aktiv auf `Low` ziehen, um dem DSP anzuzeigen, dass er den aktuellen Zugriff noch nicht abschließen kann. Dies ermöglicht eine flexible Anpassung an interne Latenzen oder Verarbeitungsschritte im FPGA – z. B. bei Pufferung oder sequenzieller Abarbeitung.

Die erreichbare Datenrate auf dem EMIFA-Bus hängt von der effektiven Zeit pro übertragenem Wort ab. Diese setzt sich aus der Dauer eines vollständigen EMIFA-Zyklus sowie eventuellen zusätzlichen Leerlaufphasen zusammen:

$$t_{\text{eff}} = t_{\text{EMIFA_Zyklus}} + t_{\text{P}} \quad (2.20)$$

$$D = \frac{\text{Wortbreite [bit]}}{t_{\text{eff}} [\text{ns}]} \quad (2.21)$$

Der EMIFA-Zyklus umfasst alle fest definierten Phasen eines asynchronen Lese- oder Schreibvorgangs (Setup, Strobe, Hold, Turn-Around bzw. Wait-Zyklen). Die zusätzliche Leerlaufphase t_{P} ergibt sich aus der Art, wie der EMIFA angesteuert wird: Bei DSP-Core gesteuerten Zugriffen entstehen zwischen zwei Transfers merkbare Idle-Intervalle, weil der DSP-Kore den nächsten Schreib/Lese-Zugriff nicht kontinuierlich nachschiebt. Wird der EMIFA dagegen vom EDMA betrieben, entfallen diese Pausen weitgehend, sodass $t_{\text{eff}} \approx t_{\text{EMIFA_Zyklus}}$ und damit deutlich höhere praktische Datenraten erreicht werden (siehe Abschnitt 5.3.1).

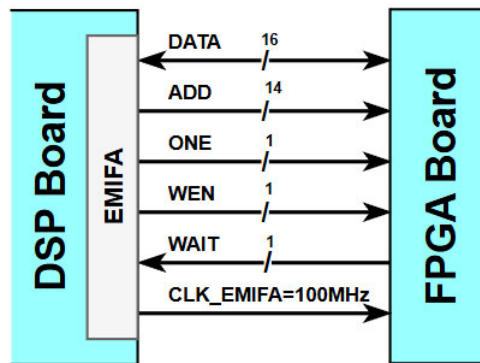


Abbildung 2.16: Signalverbindung zwischen DSP und FPGA über den EMIFA-Bus

Abbildung 2.16 zeigt den Signalfluss zwischen DSP und FPGA. Der EMIFA kann dabei mit einer Taktrate von bis zu **100 MHz** betrieben werden, was – je nach konfigurierter Buslatenz – eine hohe Datenrate ermöglicht. Die tatsächliche Performance hängt jedoch wesentlich von den eingestellten Timing-Parametern und der Implementierung des EMIFA-Interfaces im FPGA ab.

2.5.2 Rolle des EMIFA im DSP–FPGA-Codesign

Im Rahmen des Hardware-Software-Codesigns bildet der EMIFA-Bus das Bindeglied zwischen dem softwarebasierten DSP-Teil (z. B. Steuerung, Kontrollfluss, Koordination) und den hardwarebeschleunigten Signalverarbeitungsblöcken im FPGA. Die Performance dieser Schnittstelle ist daher sehr wichtig für das Gesamtsystem: Ein langsamer oder blockierender Datentransfer würde die Vorteile der parallelen FPGA-Verarbeitung stark einschränken.

Daher ist es ein Ziel dieser Arbeit, die EMIFA-Kommunikation so zu konfigurieren, dass sie mit minimaler Latenz und maximaler Übertragungsrate arbeitet. Wie diese Konfiguration konkret umgesetzt und bewertet wird, ist im Kapitel 5.3.1 ausführlich beschrieben.

2.5.3 EDMA im DSP

Zur effizienten Datenübertragung zwischen Speicherbereichen oder Peripherieeinheiten – etwa zwischen dem DSP und dem FPGA über EMIFA – stellt der TMS320C6747 einen **Enhanced Direct Memory Access (EDMA)**-Controller bereit [4]. EDMA erlaubt es, Datenblöcke autonom – d. h. ohne direkte Beteiligung der DSP-Core – zu verschieben und entlastet den Prozessor erheblich bei zeitkritischen oder wiederkehrenden Transfers.

Funktionsprinzip: Anstatt dass der DSP-Core einzelne Datenworte liest oder schreibt, wird ein *Transfer-Request* einmalig konfiguriert. Der EDMA-Controller übernimmt anschließend den kompletten Datenfluss zwischen Quelle und Ziel. Währenddessen kann der DSP andere Aufgaben verarbeiten.

Wichtige Merkmale:

- **Autonom:** Keine DSP-Core Intervention während der Übertragung notwendig.
- **Parallel:** Bis zu 32 Kanäle, die unabhängig oder synchronisiert arbeiten können.
- **Flexibel:** Unterstützt Einzeltransfers und Blocktransfers.
- **Interruptbasiert:** Transferabschluss kann optional einen Interrupt auslösen.

Einsatz im Codesign: Im Rahmen des DSP-FPGA-Codesigns wird EDMA genutzt, um größere Datenmengen effizient über den EMIFA-Bus in Richtung FPGA oder zurück in den DSP-Speicher zu übertragen – ohne dabei Taktzyklen der DSP-Core zu verschwenden.

2.6 AXI4-Stream Bus

Im Gegensatz zu speicherbasierten Bussen wie dem EMIFA setzt der **AXI4-Stream** auf ein reines *Datenstromprinzip*, das besonders gut für kontinuierliche, latenzarme Datenverarbeitung geeignet ist [13]. Das Protokoll stammt aus der AXI4-Familie (Advanced eXtensible Interface) von ARM/AMD, wird aber speziell für den unidirektionalen Datentransfer ohne Adress- und Steuerphase verwendet – z. B. zwischen Modulen in HLS-basierten Designs oder innerhalb von FPGA-Datenpfaden.

2.6.1 Funktionsweise und Signalübersicht

Der AXI4-Stream ist ein minimalistisches Protokoll mit klar definierten Steuersignalen, das auf einfache Weise synchrone Punkt-zu-Punkt-Datenströme abbildet. Die Kommunikation erfolgt taktbasiert und ohne Adressierung – lediglich über ein simples Handshake-Verfahren:

- **TDATA** – Nutzdatenbus (z. B. 16, 32, 64 oder 128 Bit).
- **TVALID** – Signal vom Sender: zeigt an, dass **TDATA** gültige Daten enthält.
- **TREADY** – Signal vom Empfänger: zeigt an, dass neue Daten angenommen werden können.
- **TLAST** – Markiert das Ende eines Rahmens oder Pakets.

Zusätzliche optionale Signale können bei Bedarf verwendet werden:

- **TKEEP / TSTRB** – Byte-Masken zur selektiven Gültigkeitsanzeige bei breiten Datenbussen.
- **TUSER** – Zusatzinformationen wie Fehlerflags oder Kanalnummern.
- **TID, TDEST** – Paketkennung bzw. Zielrouting für komplexere Datenpfade.

Die Datenübertragung basiert auf einem einfachen **Handshake-Mechanismus**: Eine Datenübertragung erfolgt nur, wenn sowohl **TVALID** (vom Sender) als auch **TREADY** (vom Empfänger) auf **High** sind. Ist eines der Signale auf **Low**, wird der Transfer pausiert. Dies erlaubt dem Empfänger, bei internen Engpässen den Datenfluss temporär zu stoppen – ein Konzept, das als *Backpressure* bezeichnet wird.

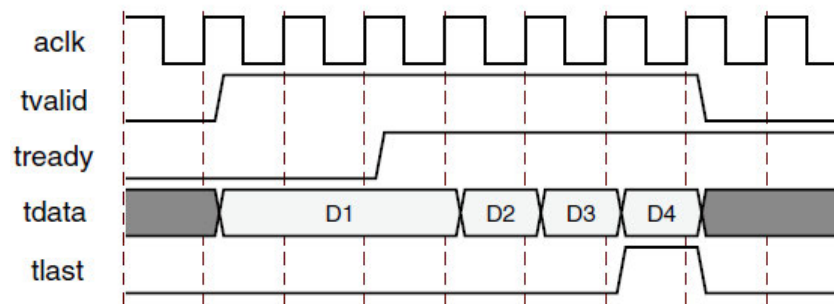


Abbildung 2.17: Signalstruktur des AXI4-Stream-Interfaces [13]

Abbildung 2.17 zeigt den Ablauf einer typischen AXI4-Stream-Datenübertragung. Sobald das sendende Modul das Signal **TVALID** auf **High** setzt, werden gültige Daten auf dem **TDATA**-Bus bereitgestellt. Die Übertragung findet jedoch erst dann statt, wenn das empfangende Modul durch **TREADY** = **High** seine Bereitschaft signalisiert. Sobald beide Signale gleichzeitig aktiv sind, wird ein gültiges Datenwort mit der nächsten Taktflanke übertragen. Das letzte Datenwort eines Transfers wird durch ein gesetztes **TLAST**-Signal markiert. Nach dessen Übertragung ist der aktuelle Datenstrom abgeschlossen.

Leistungsperspektive:

Bei dauerhaft **TVALID**=1 und **TREADY**=1 kann pro Takt ein komplettes Datenwort übertragen werden. Damit erlaubt AXI4-Stream bei hoher Taktfrequenz (z. B. 100 MHz) eine extrem hohe Datenrate bei gleichzeitig sehr geringer Latenz – ideal für die Kopplung von HLS-Funktionsblöcken wie FFT, Modulatoren oder Decodern [28].

Die hier vorgestellte AXI4-Stream-Schnittstelle spielt eine zentrale Rolle im späteren Hardware-Software-Codesign: Sie erlaubt es, rechenintensive HLS-IP-Cores flexibel mit FIFO-Interfaces oder direkt mit anderen Stream-Modulen zu verbinden. Die konkrete Anbindung von HLS-IP-Cores über AXI4-Stream erfolgt in den Kapiteln 5.4.2 und 5.4.2, in denen die Umsetzung von FIFO-zu-AXI- und AXI-zu-FIFO-Interfaces beschrieben wird.

3 Anforderungsanalyse

3.1 Einleitung

Dieses Kapitel beschreibt die fachlichen und technischen Anforderungen an ein OFDM-basiertes Demonstratorsystem auf einer heterogenen DSP-FPGA-Plattform. Das zentrale Ziel ist die Bewertung einer HLS-gestützten Hardware-Software-Codesign-Methodik. OFDM dient dabei als exemplarische, kontinuierliche Streaming-Anwendung, um den Ansatz praxisnah zu demonstrieren und quantitativ zu bewerten. Die Anforderungen wurden in Abstimmung mit dem Modulverantwortlichen des Moduls *Embedded Signal Processing* definiert und im Projektverlauf iterativ konkretisiert. Jede Anforderung ist einer Priorität (P1 = hoch, P2 = mittel, P3 = niedrig) zugeordnet und mit klaren Akzeptanzkriterien versehen. Dadurch lässt sich die Zielerreichung transparent bewerten und den Ergebnissen aus Systemarchitektur, Implementierung und Verifikation eindeutig zuordnen.

Betrachtet wird der Basisband-Datenpfad einschließlich der Plattformintegration, insbesondere die EMIFA-Schnittstelle zwischen DSP und FPGA, die Entwicklung der Signalverarbeitungsmodule in HLS, Audio-I/O sowie DMA-/Interrupt-Mechanismen. Nicht Bestandteil sind Hochfrequenz-Frontend, Protokolle oberhalb einfacher Rahmenbildung, EMV- und Sicherheitsaspekte. Die gewählten Kennzahlen sind so definiert, dass sie über den konkreten OFDM-Anwendungsfall hinaus auf vergleichbare Signalverarbeitungsaufgaben übertragbar sind.

3.2 Ziele und Vorgehen

Dieser Abschnitt fasst die Projektziele zusammen und beschreibt das Vorgehen zur Umsetzung. Der Schwerpunkt liegt auf der Untersuchung einer HLS-gestützten Hardware-Software-Codesign-Methodik auf einer DSP-FPGA-Plattform.

Projektziele

Im Rahmen dieses Projekts wurden konkrete Teilziele definiert, um die Eignung von High-Level-Synthese im Hardware-Software-Codesign anhand eines OFDM-Systems zu untersuchen. Die Prioritäten der Ziele wurden in Abstimmung mit dem Modulverantwortlichen des Moduls *Embedded Signal Processing* definiert und dienen der Fokussierung auf funktional und methodisch relevante Aspekte der Umsetzung.

- **Z1:** Aufbau einer stabilen EMIFA-Datenanbindung auf FPGA-Seite und Sicherstellung einer zuverlässigen Kommunikation zwischen DSP und FPGA als Grundlage für den Gesamtsystembetrieb im Codesign.
- **Z2:** Umsetzung einer angepassten OFDM-Sende- und Empfangskette zur exemplarischen Abbildung einer realitätsnahen Signalverarbeitung im Basisband.
- **Z3:** Entwicklung eines MATLAB-Referenzmodells zur Simulation der Sende- Empfangsstrecke und zum funktionalen Abgleich der auf FPGA (VHDL/HLS) und DSP (C) umgesetzten Verarbeitungsschritte.
- **Z4:** Analyse und Bewertung unterschiedlicher Aufgabenteilungen zwischen DSP und FPGA mit Fokus auf Rechenlast, Parallelisierbarkeit und Steuerlogik.
- **Z5:** Ermöglichung einer flexiblen Parametrierung zentraler Systemkomponenten (z. B. FFT-Länge, CP-Anteil, Modulationsordnung) zur Untersuchung von Varianten sowie zur Wiederverwendbarkeit in künftigen Projekten. Eine Umschaltung zur Laufzeit ist nicht vorgesehen.
- **Z6:** Integration aller Systemkomponenten zu einer durchgängigen OFDM Verarbeitungskette mit vollständigem Funktionsnachweis.
- **Z7:** Einsatz von High-Level-Synthese zur Implementierung zentraler Verarbeitungsmodule mit anschließender Bewertung hinsichtlich Effizienz, Ressourcennutzung und Integrationsfähigkeit im Codesign-Kontext.

Vorgehen

Das folgende Vorgehensmodell leitet sich direkt aus den zuvor definierten Projektzielen ab und beschreibt die Umsetzungsschritte zur systematischen Realisierung und Bewertung

des OFDM-Demonstrators. Die einzelnen Phasen sind dabei so strukturiert, dass sie zentrale Anforderungen wie die FPGA/DSP-Integration (Z1, Z4), die HLS-Implementierung (Z7) sowie die Parametrierbarkeit und Verifikation (Z3, Z5, Z6) gezielt adressieren. Alle Schritte bauen logisch aufeinander auf und ermöglichen eine schrittweise Validierung der entworfenen Codesign-Architektur.

1) Definition einer Startkonfiguration. Zur Absicherung reproduzierbarer Tests und effizienter Entwicklungszyklen wird eine reduzierte Ausgangskonfiguration mit festen Parametern festgelegt. Diese umfasst z.B. NFFT=64, ein CP-Verhältnis von 1/8, 16QAM-Modulation und blockweise Verarbeitung. Als I/O-Schnittstelle werden Datei- oder Speicherpfade genutzt; Audio-I/O ist optional.

2) Erstellung des MATLAB-Referenzmodells. Implementierung eines vollständigen Simulationsmodells der OFDM-Sende- und Empfangsstrecke zur Generierung von Testdaten und zum Abgleich mit den Hardware-/Software-Komponenten. Der Übertragungskanal wird im MATLAB-Referenzmodell als AWGN-Modell realisiert.

3) Inbetriebnahme der EMIFA-Schnittstelle. Einrichtung der Register- und Datenpfade für die Kommunikation zwischen DSP und FPGA. Es werden DMA-basierte sowie wortweise Transfers umgesetzt und unter definierten Lastprofilen getestet. Die Performance (Durchsatz, Stabilität, Timing) wird dokumentiert.

4) Partitionierung und Schnittstellendefinition. Analyse der Signalverarbeitungskette im Hinblick auf Rechenlast, Parallelisierbarkeit und Kontrollbedarf. Daraus wird eine funktionale Aufteilung auf DSP und FPGA abgeleitet. Datenformate, Adressbereiche und Steuermechanismen werden konkret festgelegt.

5) Entwicklung der HLS-Module. Rechenintensive Blöcke wie IFFT/FFT, CP Verarbeitung sowie Mapper/Demapper werden in HLS implementiert. Dabei werden systematisch verschiedene HLS-Pragmas (z. B. pipeline, dataflow, unroll) auf Architektur und Performanz untersucht.

6) Integration auf DSP und FPGA. Implementierung der Ansteuerungslogik auf beiden Seiten zur bidirektionalen Datenübertragung über EMIFA. Auf dem DSP erfolgt die Steuerung und Sequenzierung, auf dem FPGA die Einbindung der HLS-Module in die Verarbeitungskette. Alle Schnittstellen und Abläufe werden umfassend getestet und dokumentiert.

7) Verifikation und Ergebnisdokumentation. Funktionstest der Einzelmodule mit VHDL- und C-Testbenches sowie schrittweiser Abgleich des Gesamtsystems mit dem MATLAB-Modell. Alle wesentlichen Entwicklungsschritte werden in Form von Blockdiagrammen, Schnittstellenbeschreibungen, Build-Prozesse und Testergebnisse nachvollziehbar dokumentiert.

3.3 Funktionale Anforderungen

In diesem Abschnitt wird festgelegt, welche funktionalen Leistungen das System im Rahmen der Umsetzung erbringen muss. Die Anforderungen ergeben sich aus den Zielen Z1 bis Z7, den spezifischen Eigenschaften der eingesetzten DSP-FPGA-Plattform sowie dem Anwendungsszenario einer signaltechnischen Verarbeitungskette zur Übertragung von Audiodaten über ein OFDM-System.

- **F1 (P1) OFDM-Basisbandverarbeitung:** Die OFDM-Verarbeitungskette im reellen Basisband umfasst die wesentlichen Funktionsblöcke für Mapping, IFFT/FFT, das Einfügen bzw. Entfernen des Cyclic Prefix sowie Demapping. Sender- und Empfängerpfad sind auf derselben DSP-FPGA-Plattform zu implementieren. Die funktionale Korrektheit wird durch den Vergleich mit Referenzvektoren aus einem MATLAB-Modell überprüft, wobei ein vereinfachter Übertragungskanal mit additivem Weißrauschen (AWGN) angenommen wird.
- **F2 (P1) EMIFA-Datenkommunikation zwischen DSP und FPGA:** Die Daten- und Registerschnittstelle über EMIFA wird stabil betrieben. Die Funktionsfähigkeit wird über Integrations- und Lasttests mit protokollierten Transfers sowie die Sichtprüfung der Register- und Datenpfade nachgewiesen.
- **F3 (P1) MATLAB-Referenzmodell:** Ein Goldmodell wird entwickelt und dient dem funktionalen Abgleich der Implementierungen. Seine Eignung wird durch Modellreview und erfolgreiche Verwendung der Vektoren in den Tests belegt.
- **F4 (P1) Ende-zu-Ende-Verarbeitung von Audiodaten:** Ein Audiosignal wird vollständig über die implementierte OFDM-Verarbeitungskette übertragen. Übertragungsbedingte Verzerrungen dürfen dabei nicht über unvermeidbare Quantisierungseffekte und modelliertes Rauschen (z. B. AWGN) hinausgehen. Die Signalqualität wird durch Wellenformvergleiche, Spektralanalysen und objektive Metriken

wie das segmentelle SNR bewertet. Eine ergänzende Hörprobe dient der Plausibilisierung.

- **F5 (P2) Flexible Parametrierung:** Zentrale Systemparameter wie FFT-Länge, Cyclic-Prefix-Anteil oder Modulationsordnung können in der Implementierung angepasst werden. Die korrekte Funktionsweise bei unterschiedlichen Konfigurationen wird durch systematische Tests und reproduzierbare Ergebnisse überprüft.
- **F6 (P1) Digitaler Modulator/Demodulator:** Ein digitaler Modulator und Demodulator (z. B. für 16-QAM) ist Bestandteil der Verarbeitungskette. Die korrekte Funktion wird durch den Vergleich der demodulierten Symbolfolgen mit Referenzdaten sowie durch die visuelle Auswertung der Konstellationsdiagramme nachgewiesen. Dabei wird die erwartete Symbolverteilung gegen die dekodierten Punkte aus der Implementierung geplottet und bewertet.
- **F7 (optional) Vorwärtsfehlerkorrektur:** Zur optionalen Erweiterung kann ein einfacher FEC-Mechanismus (z. B. Faltungscodes mit Viterbi-Decoder) in die Verarbeitungskette eingebunden werden. Seine Funktion wird anhand synthetischer Testsequenzen und dem Abgleich mit einer Referenzdecodierung bestätigt.
- **F8 (P3) Datenübertragung und Pufferverwaltung auf dem DSP:** Zur Kommunikation mit dem FPGA werden auf dem DSP sowohl wortweise als auch blockweise Übertragungsmodi realisiert. Die korrekte Funktion, insbesondere die Taktung und Pufferumschaltung, wird durch Signalverfolgung an ausgewählten EMIFA-Leitungen mittels Oszilloskop überprüft.
- **F9 (P3) Quellenkodierung (optional):** Zur gezielten Belastung der Datenpfade kann eine einfache Quellenkodierung für Audiodaten ergänzt werden. Ihre Funktionalität wird anhand von Beispielmateriale überprüft, indem die dekodierten Daten mit dem Originalsignal verglichen werden.

Die Überprüfung der funktionalen Anforderungen erfolgt systematisch in den nachfolgenden Kapiteln zur Implementierung und Evaluation.

3.4 Nicht-Funktionale Anforderungen

Dieser Abschnitt beschreibt die Qualitätsmerkmale des Systems. Sie ergeben sich aus den Randbedingungen der Zielplattform, der Verarbeitung von Audio-Daten über OFDM und dem methodischen Fokus auf High-Level-Synthese im Hardware-Software-Codesign.

- **NF1 (P1):** Die Rechenlast wird so zwischen DSP und FPGA aufgeteilt, dass zeitkritische Verarbeitungsschritte fristgerecht abgearbeitet und vorhandene Ressourcen effizient genutzt werden. Rechenintensive, parallelisierbare Algorithmen werden dem FPGA zugewiesen, während Kontrolllogik und sequentielle Abläufe auf dem DSP verbleiben. Die Partitionierung ist darauf ausgelegt, konstante Verarbeitungszeiten sicherzustellen und die architekturspezifischen Stärken beider Komponenten gezielt zu nutzen.
- **NF2 (P2):** Die Architektur des Gesamtsystems wird so entworfen, dass ein ausgewogenes Verhältnis zwischen Latenz, Durchsatz und Ressourcennutzung (LUTs, Flip-Flops, BlockRAM, DSP-Slices) erreicht wird. Rechenintensive Funktionen werden gezielt parallelisiert und gepipelined, wobei alle Timing-Vorgaben sicher einzuhalten sind. Darüber hinaus muss das Design stabil bei erhöhter Taktfrequenz arbeiten, ohne dass Anpassungen an Architektur oder Algorithmen erforderlich werden.
- **NF3 (P3):** Fließkomma-Algorithmen werden in geeignete Festkommadarstellungen überführt und hinsichtlich Genauigkeit, Ressourcenbedarf und Ausführungszeit bewertet. Ziel ist eine numerisch stabile, bitbreiteneffiziente Implementierung, die die Effizienzvorteile des FPGAs nutzt und gleichzeitig das modellbasierte Referenzverhalten hinreichend genau abbildet.
- **NF4 (P1):** Die HLS-Entwicklungsmethodik wird am OFDM-System nachvollziehbar dokumentiert. Dazu gehören Architekturentscheidungen, Schnittstellenkonzepte, der Einsatz von Pragmas (z. B. Pipelining, Unrolling), die Verifikation mit C/RTL-Co-Simulation und der qualitative Vergleich der erzielten Implementierungsergebnisse. Der Erkenntnisgewinn ist zentraler Bestandteil des Projekts.
- **NF5 (P2):** Mehrere FFT-Längen werden konkret auf dem Zielsystem implementiert und synthetisiert. Im Vergleich werden Latenz, Ressourcenbedarf sowie der Integrationsaufwand (Schnittstellen, Wortlängen, Pufferung, Taktdomänen) systematisch gegenübergestellt. Die für das Gesamtsystem geeignetste Variante wird

ausgewählt und so eingebunden, dass keine zusätzlichen Anpassungen an der bestehenden DSP/FPGA-Partitionierung erforderlich sind.

- **NF6 (P1):** Das System wird vollständig dokumentiert durch konsistente Blockdiagramme, Datenfluss- und Architekturbeschreibungen sowie Schnittstellen- und Parametertabellen. Dadurch werden Nachvollziehbarkeit und Wartbarkeit gesichert und bilden die Grundlage für Reviews und die Verknüpfung von Anforderungen mit Verifikationsschritten.

4 Konzept und Vorgehensweise

4.1 Einleitung

Dieses Kapitel beschreibt den Übergang von der Anforderungsanalyse zur konkreten Systemarchitektur. Auf Basis der zuvor definierten funktionalen und nichtfunktionalen Anforderungen werden hier die konzeptionellen Entwurfsentscheidungen dargestellt, die den Aufbau, die Struktur und die Kommunikationsweise des Gesamtsystems bestimmen. Im Fokus stehen die Partitionierung der Signalverarbeitung zwischen DSP und FPGA, die Definition von Datenpfaden und Taktdomänen, die Auslegung der Schnittstellen und Kommunikationsmechanismen beider Plattformen sowie die Untersuchung und Festlegung der OFDM-Parametrierung. Zudem werden die grundlegenden Prinzipien für den Einsatz von High-Level-Synthese festgelegt, mit dem Ziel, die spätere Implementierung effizient, skalierbar und ressourcenschonend zu gestalten.

Das Kapitel bildet damit die konzeptionelle Grundlage für die im folgenden Kapitel dargestellte Implementierung des OFDM-Systems.

4.2 Simulationsmodell

Zur Vorbereitung der Hardware–Software-Aufteilung wurde ein Simulationsmodell der OFDM-Sender- und Empfängerketten in MATLAB implementiert (Abbildung 4.1). Das Modell bildet die wesentlichen Verarbeitungsschritte eines OFDM-Systems ab und dient als bitgenaue Referenz zur Validierung einzelner Module im weiteren Entwicklungsprozess. Die Struktur orientiert sich dabei bewusst an einer modularen Umsetzung: Jedes Teilmodul kann separat analysiert, getestet und später gezielt für eine HLS Implementierung vorbereitet werden.

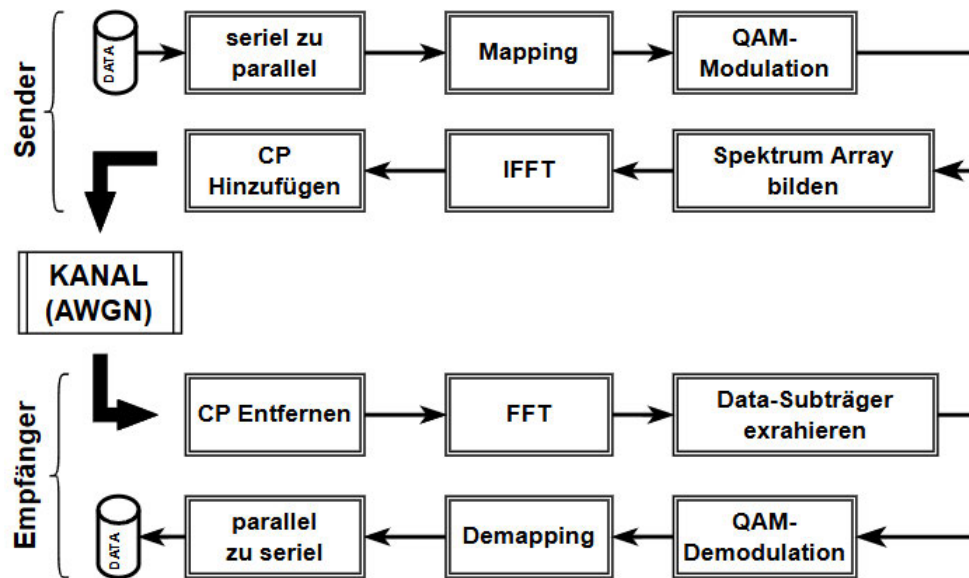


Abbildung 4.1: Blockdiagramm der MATLAB-Simulation (Tx-Rx-Kette)

Im Vergleich zur vollständigen Sender- und Empfangskette gemäß Abschnitt 2.1.2 und Abschnitt 2.1.3 enthält das MATLAB-Modell lediglich die funktional relevanten Kernblöcke für die digitale Verarbeitung. Komplexe Systemkomponenten wie Synchronisation, Kanalmodellierung oder FEC-Codierung wurden aus Gründen der Fokussierung bewusst ausgeklammert. Ziel des Modells ist nicht die Abbildung eines vollständigen OFDM-Systems, sondern die Bereitstellung einer kontrollierten Simulationsumgebung zur funktionalen Verifikation der Verarbeitungsmodule sowie zur Überprüfung der korrekten Datenpfadverknüpfung zwischen Sender- und Empfängerseite.

Das Modell umfasst konkret folgende Verarbeitungsschritte:

- Symbolmapping (16-, 64- und 256-QAM)
- Bildung des IFFT-Eingangsvektors mit Pilot- und Datensubträgern
- IFFT und Anfügen des zyklischen Präfixes (Tx)
- Entfernen des CP und FFT (Rx)
- Daten-Subträgerextraktion
- Demapping der Symbole zu Bits

Im folgenden Abschnitt werden aufbauend auf diesem Referenzmodell die eingesetzten Systemparameter, die Modulationsverfahren, das Subträgerlayout sowie abgeleitete Kenngrößen erläutert. Diese Parametrierung bildet die Grundlage für die spätere HLS-Umsetzung und das Hardware-Software-Codesign.

Randbedingungen und Parametrierung:

Zur Untersuchung des Hardware-Software-Codesigns werden drei FFT-Längen $N \in \{64, 128, 256\}$ betrachtet. Diese Werte repräsentieren realistische Konfigurationen im Hinblick auf die verfügbaren Ressourcen und Verarbeitungskapazitäten der eingesetzten Zielplattform (vgl. Abschnitt 2.4). Höhere FFT-Längen würden zwar prinzipiell eine feinere Frequenzauflösung bieten, sind jedoch mit erhöhter Latenz, größerem Speicherbedarf und höherem Ressourcenverbrauch verbunden – was im gegebenen Hardwarekontext nicht effizient umsetzbar ist [24].

In der Simulation wird reelles OFDM mit hermitischer Symmetrie verwendet, da die Zielanwendung eine Audioausgabe erfordert. Aus diesem Grund werden nur Subträger im positiven Frequenzbereich mit Nutzdaten belegt. Die Subträger im negativen Spektralbereich werden durch komplex konjugierte Spiegelung erzeugt, sodass das resultierende IFFT-Signal im Zeitbereich reellwertig bleibt. Der Gleichanteil (DC-Komponente) sowie der Subträger bei der Nyquist-Frequenz bleiben ungenutzt – wie in Abschnitt 2.1.2 beschrieben – da sie keine nutzbare Information transportieren und typischerweise im OFDM-Spektrum auf Null gesetzt werden. Zusätzlich werden an den Bandrändern Guardträger auf Null gesetzt, um Störungen in benachbarten Frequenzbändern zu vermeiden [44] [32].

Der Übertragungskanal wird als reiner **AWGN-Kanal** modelliert [7], d. h. es wird ausschließlich additives weißes Gaußsches Rauschen hinzugefügt. Zeitvarianz, Frequenzselektion, Doppler-Effekte oder Mehrwegeausbreitung werden explizit nicht berücksichtigt, da der Fokus dieser Arbeit auf der Hardware-Umsetzung liegt und nicht auf der Optimierung des Kommunikationssystems im physikalischen Sinne. Durch diese Vereinfachung kann der Einfluss unterschiedlicher FFT-Längen auf Pufferung, Latenz und Ressourcenverbrauch untersucht werden, ohne dass kanalselektive Effekte das Verhalten verfälschen.

Die Subträgeranzahl für Nutzdaten, Piloten und Nullträger wächst proportional zur FFT-Länge. Tabelle 4.1 zeigt die verwendete Subträgeraufteilung. Die Bruttodatenrate bleibt bei konstanter Abtastrate $f_s = 100 \text{ kHz}$ über alle Konfigurationen hinweg identisch, da mit $\Delta f = f_s/N_{\text{FFT}}$ das Produkt $N_{\text{Daten}} \cdot \Delta f$ konstant bleibt. Die Wahl von f_s basiert

auf der angestrebten Echtzeitverarbeitung und ist auf die Audioanwendung und die Leistungsfähigkeit der DSP/FPGA-Plattform abgestimmt. Die Unterschiede zwischen den untersuchten FFT-Längen wirken sich somit primär auf die Verarbeitungsarchitektur (Latenz, Speicherbedarf, Parallelität) aus, nicht auf die Übertragungsrate.

N	N_{Daten}	N_{Pilot}	N_{Null}
64	24	4	DC/Nyquist + Guards = 4
128	48	8	DC/Nyquist + Guards = 8
256	96	16	DC/Nyquist + Guards = 16

Tabelle 4.1: Subträgerbelegung je FFT-Länge (reelles OFDM).

Modulationsauswahl:

Basierend auf den in Abschnitt 2.1.3 beschriebenen Grundlagen ist QAM für OFDM-Systeme besonders gut geeignet [18] [14]. Sie bietet eine hohe spektrale Effizienz und ist technisch gut in I/Q-basierte Übertragungen integrierbar. Zur Bewertung verschiedener Modulationsordnungen wurden für $M \in \{16, 64, 256\}$ sowohl die theoretischen Bitfehlerraten (BER) im AWGN-Kanal als auch die BER-Kurven aus der MATLAB-basierten OFDM-Simulation berechnet. Dabei wurden jeweils FFT-Längen $N \in \{64, 128, 256\}$ berücksichtigt. Ein Beispiel für $N = 128$ ist in Abbildung 4.2 dargestellt, die übrigen Kurven befinden sich im Anhang A.1.

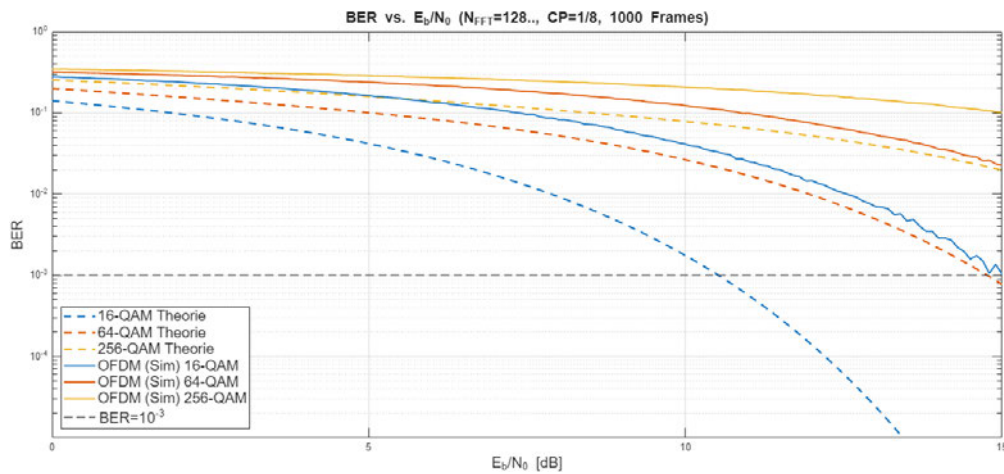


Abbildung 4.2: BER gegen E_b/N_0 für 16/64/256-QAM: Theorie (AWGN) und OFDM-Simulation mit $N = 128$.

In der Simulation wird die Energie pro Bit E_b nicht isoliert für die Nutzdaten berechnet, sondern aus der Gesamtenergie eines OFDM-Rahmens abgeleitet – einschließlich Pilotsignalen und zyklischem Präfix. Dadurch liegt E_b unter dem theoretischen Wert, was zu etwas höheren BER-Werten führt. Dieser Ansatz ist realitätsnah und bildet den Systemaufwand realistischer ab.

Als Orientierung für die Bewertung dient ein BER-Wert von etwa 10^{-3} . Diese Schwelle wird häufig verwendet, da sie in der Praxis bereits eine brauchbare Übertragungsqualität liefert. Durch einfache Fehlerkorrekturverfahren wie Faltungscodes kann sie zusätzlich deutlich verbessert werden.

Die Ergebnisse zeigen, dass 16-QAM über alle untersuchten FFT-Größen hinweg die robusteste Leistung bietet. Sie erreicht den Ziel-BER-Wert bei vergleichsweise geringem Energiebedarf und stellt geringere Anforderungen an die Signalqualität. Höhere Modulationsstufen wie 64- oder 256-QAM liefern zwar eine höhere Datenrate, erfordern aber auch ein höheres E_b/N_0 , eine präzisere Abtastung und eine deutlich feinere Entscheidungsgenauigkeit im Empfänger.

Wie im Abschnitt 2.1.3 veranschaulicht wurde, je höher die Modulationsordnung M , desto dichter liegen die einzelnen Symbole im zweidimensionalen I/Q-Diagramm beieinander. Der Empfänger muss anhand der empfangenen (verrauschten) Signalpunkte entscheiden, welchem idealen QAM-Symbol sie am nächsten liegen – also auf welches Bitmuster sie abgebildet werden. Bei 256-QAM gibt es 256 mögliche Symbolpositionen in einem Gitterraster. Bereits kleine Störungen durch Rauschen, Jitter oder Nichtlinearitäten können dazu führen, dass ein empfangenes Symbol dem falschen Entscheidungsgitter zugeordnet wird. Dadurch steigt die Fehleranfälligkeit bei gleicher Kanalqualität deutlich an [58]. Diese sogenannte *Entscheidungsverschärfung* macht höhere QAM-Ordnungen besonders empfindlich gegenüber Störungen.

Da die Nutzdaten in einem realen System durch einen ADC blockweise zur Verarbeitung bereitgestellt werden, erhöht sich mit steigender Modulationsordnung auch die Datenmenge, die benötigt wird, um ein vollständiges OFDM-Symbol zu füllen (siehe Abschnitt 2.4). Bei 256-QAM müssen beispielsweise acht Bits pro Symbolstelle gesammelt werden, bevor die IFFT gebildet werden kann. Diese verlängerte *Füllzeit* wirkt sich unmittelbar auf die Gesamtlatenz des Systems aus. Unter *Latenz* wird hier die gesamte Verzögerung verstanden, bis ein vollständiger IFFT-Eingangsblock mit allen benötigten Bits zur Verfügung steht – einschließlich Puffer- und Verarbeitungsschritte im DSP- und FPGA-Pfad.

Tabelle 4.2 zeigt, wie viele 16-Bit-Nutzdatenwörter sich bei verschiedenen FFT-Längen und Modulationsordnungen pro OFDM-Symbol übertragen lassen. Höhere Modulationsordnungen steigern zwar den Durchsatz, führen aber zu höherem Energiebedarf und erhöhter Empfindlichkeit gegenüber Störungen. Diese Tabelle hilft dabei, den Zusammenhang zwischen Symbolkapazität, Systemkomplexität und Fehlerrate greifbar zu machen.

N	N_{Daten}	16-QAM	64-QAM	256-QAM
64	24	6	9	12
128	48	12	18	24
256	96	24	36	48

Tabelle 4.2: 16-Bit-Nutzdatenwörter pro OFDM-Symbol (nur Datenträger) für verschiedene N und QAM-Ordnungen.

Subträgerlinien und abgeleitete Kenngrößen:

Die spektrale Struktur eines OFDM-Signals ergibt sich – wie in Abschnitt 2.1.2 beschrieben – aus der Überlagerung vieler Einzelsignale in Form von Subträgern [18]. Jeder Subträger trägt einen Teil der Information und ist in einem genau festgelegten Frequenzabstand zur nächsten Trägerfrequenz positioniert. Abbildung 4.3 zeigt exemplarisch das Frequenzspektrum bei $N = 128$. Die durchgezogenen Linien stellen aktive **Datenträger** dar, die hellblau gefüllten Linien sind **Pilotträger**, welche für die spätere Kanalabschätzung vorgesehen sind. Zwischen diesen Subträgern sind **Guardzonen** erkennbar – also bewusst freigelassene Frequenzbereiche am Rand des nutzbaren Spektrums. Diese dienen dazu, benachbarte Kanäle nicht zu stören [25]. Auch die Frequenzposition bei Null (**DC-Träger**) und bei der höchsten darstellbaren Frequenz (**Nyquist-Träger**, $f = f_s/2$) bleibt leer. Dies ist typisch für reelles OFDM mit hermitischer Symmetrie und wurde bereits in Abschnitt 2.1.2 eingeführt: Damit das Ausgangssignal rein reell ist, müssen negative und positive Frequenzanteile symmetrisch sein – was nur gelingt, wenn bestimmte Träger nicht belegt werden.

Auffällig in der Darstellung ist, dass die Subträger nicht alle gleich hoch sind. Das liegt daran, dass keine künstliche Pegelnormalisierung vorgenommen wurde. Die Höhe der einzelnen Linien hängt vom verwendeten Modulationssymbol ab. Bei 16-QAM beispielsweise kann jeder Subträger verschiedene Amplituden annehmen, abhängig von der übertragenen Bitkombination. Manche Symbole haben dieselbe Amplitude, aber eine unterschiedliche Phase – diese sind im Spektrum zwar gleich hoch, unterscheiden sich aber im zeitlichen Signalverlauf.

Ein weiterer Effekt wird bei wachsendem N sichtbar: Der Abstand $\Delta f = f_s/N$ zwischen den Subträgern wird kleiner, wodurch die Linien im Frequenzbereich näher zusammenrücken. Dies erhöht zwar die spektrale Effizienz, kann aber die Empfindlichkeit gegenüber Frequenzfehlern oder Doppler-Effekten steigern, da sich benachbarte Subträger stärker beeinflussen. Solche Effekte sind im hier verwendeten AWGN-Modell nicht berücksichtigt, werden aber in realen Funkkanälen relevant.

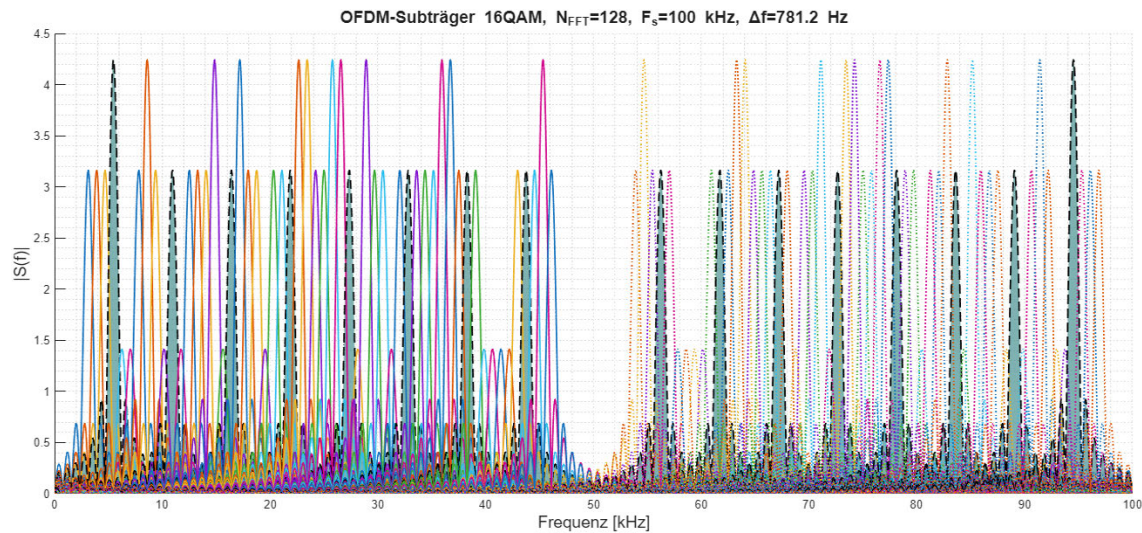


Abbildung 4.3: Subträgerstruktur bei $N = 128$ mit Pilot- und Datenträgern (reelles OFDM, $f_s = 100$ kHz).

Die zentralen Kenngrößen eines OFDM-Symbols lassen sich direkt aus der FFT-Größe und der Abtastrate $f_s = 100$ kHz ableiten. Tabelle 4.3 zeigt den Subträgerabstand Δf , die Nutzsymboldauer T_u , die Dauer des zyklischen Präfixes T_{CP} sowie die Gesamtsymboldauer $T_{\text{sym}} = T_u + T_{CP}$. Als Verhältnis wurde $T_{CP} = T_u/8$ gewählt, da dieses Verhältnis in typischen Mehrwegekanälen einen ausreichenden Schutz vor Intersymbolinterferenzen bietet [35].

N	Δf [Hz]	T_u [ms]	T_{CP} [ms]	T_{sym} [ms]
64	1562.5	0.64	0.08	0.72
128	781.25	1.28	0.16	1.44
256	390.625	2.56	0.32	2.88

Tabelle 4.3: Abgeleitete Kenngrößen bei $f_s = 100$ kHz und $CP = 1/8$.

Je größer also die FFT-Länge, desto länger dauert ein OFDM-Symbol, bei gleichbleibender Abtastrate. Das wirkt sich unmittelbar auf die Systemlatenz aus und muss bei der späteren Hardware-Umsetzung berücksichtigt werden.

Zeitsignal:

Das OFDM-Zeitsignal entsteht durch die IFFT der belegten Subträger, wie in Abschnitt 2.1.4 beschrieben [57]. Im resultierenden Signal sind alle Informationen eines Symbols zeitlich zusammengefasst. Dieses Signal bildet die physikalische Basis der Übertragung – es ist das, was vom Sender ausgestrahlt und vom Empfänger wieder aufgenommen wird.

Abbildung 4.4 zeigt exemplarisch ein solches Zeitsignal für $N = 128$. Der gelb markierte Abschnitt stellt den CP dar, der dem OFDM-Symbol vorangestellt wird. Dieser CP dient dazu, die Auswirkungen von Laufzeitunterschieden im Kanal abzufangen und die Orthogonalität der Subträger zu erhalten – eine zentrale Voraussetzung für störungsfreie Demodulation im Empfänger.

Typisch für OFDM ist, dass das Zeitsignal relativ gleichmäßige Mittelwerte, aber einzelne hohe Leistungsspitzen aufweist. Diese Amplitudenspitzen – sichtbar als scharfe Ausschläge im Zeitverlauf – sind Ausdruck der hohen *PAPR*.

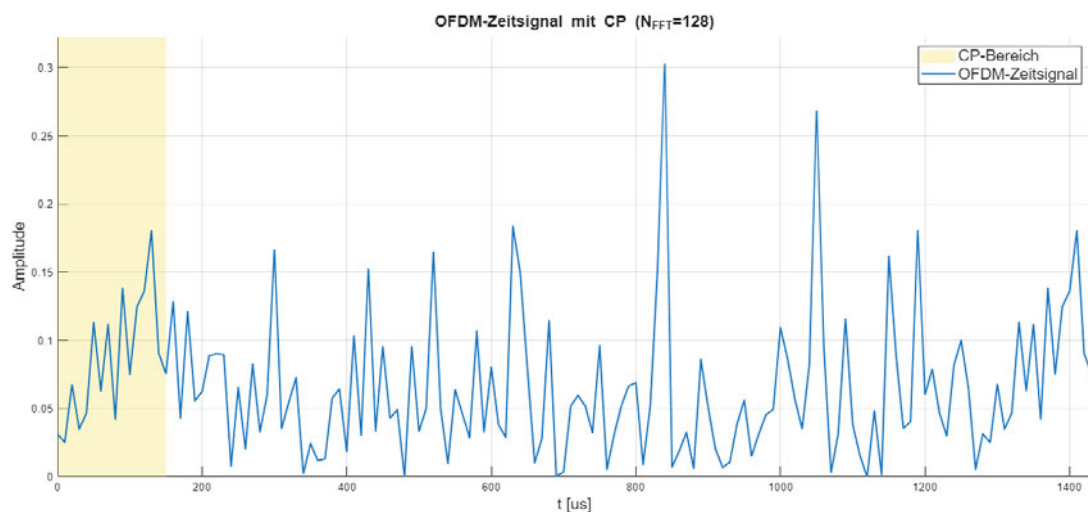


Abbildung 4.4: OFDM-Zeitsignal mit CP (gelb markiert), $N = 128$.

In realen Systemen können diese Leistungsspitzen die Auslegung des Leistungsverstärkers erheblich beeinflussen. Eine Reduktion der PAPR ist zwar möglich, z. B. durch Verfahren wie Selected Mapping (SLM) [57], bei denen mehrere phasenverschobene Varianten eines

Symbols erzeugt werden. Diese Option wurde im Rahmen dieser Arbeit jedoch nicht weiterverfolgt. Die hier verwendeten Zeitsignale stammen direkt aus der IFFT, ohne zusätzliche PAPR-Optimierung.

Leistungsdichtespektrum (PSD):

Neben dem Zeitsignal ist auch das Frequenzverhalten eines OFDM-Signals für viele Anwendungen von Interesse – z. B. bei der Einhaltung von Bandbreitenvorgaben oder der Filterauslegung bei der späteren Hardwareintegration [57] [53]. Daher wurde ergänzend eine spektrale Analyse durchgeführt, um zu zeigen, wie sich die Belegung der Subträger im Frequenzbereich äußert und welchen Einfluss die FFT-Größe auf das Spektrum hat.

Dazu wurde eine lange OFDM-Zeitfolge synthetisch erzeugt – bestehend aus 1000 aufeinanderfolgenden Symbolen mit $N \in \{64, 128, 256\}$, jeweils ohne zyklischen Präfix. Die Zeitreihe wurde auf mittlere Leistung normiert.

Zur Schätzung des Spektrums wurde die Welch-Methode mit Hann-Fensterung verwendet [18]. Abbildung 4.5 zeigt die berechnete spektrale Leistungsdichte (PSD) für alle drei FFT-Größen. Deutlich zu erkennen ist, dass das belegte Frequenzband über alle Varianten gleich breit bleibt – ein erwartetes Verhalten, da die Abtastrate konstant ist.

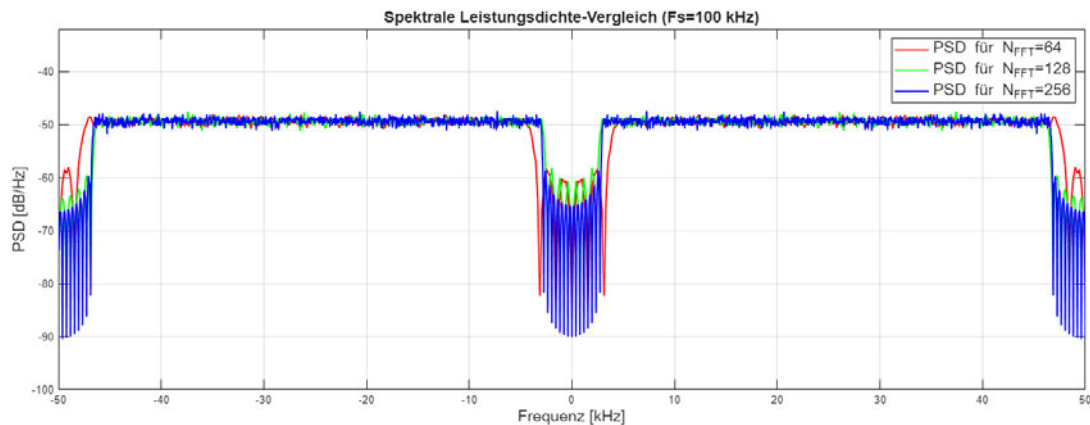


Abbildung 4.5: PSD-Vergleich für $N_{\text{FFT}} = 64, 128, 256$ bei $f_s = 100$ kHz.

Zusätzlich fallen die spektralen Lücken bei DC (0 Hz) und bei der Nyquist-Frequenz ($f = f_s/2$) auf. Diese ergeben sich durch die reelle Signalstruktur mit hermitischer Symmetrie (siehe Abschnitt 2.1.2). Auch die äußeren Guardbänder sind sichtbar: Sie verhindern, dass die OFDM-Spektren in benachbarte Frequenzbereiche ausstrahlen – ein typisches Designmerkmal in realen Systemen [58].

Die gezeigte Darstellung dient in dieser Arbeit vor allem der Veranschaulichung der spektralen Eigenschaften und als Referenz für mögliche spätere Optimierungsschritte (z. B. Fensterung oder Bandbegrenzung). Eine tiefergehende spektrale Optimierung ist nicht Gegenstand dieser Arbeit.

Standardbezug:

Das entwickelte OFDM-System orientiert sich in Aufbau und grundlegender Parametrierung an den Standards wie IEEE 802.11a und LTE. Für die Umsetzung auf der verfügbaren Hardwareplattform wurden jedoch Anpassungen vorgenommen – insbesondere bei Symbolrate, FFT-Länge und Subträgerbelegung –, um die Anforderungen an Ressourcen, Taktfrequenz und Echtzeitfähigkeit zu erfüllen [20] [35].

Anforderungsbezug und Methodik:

Das Simulationsmodell erfüllt wichtige Funktionen im Projekt: Es dient als Referenz zur Verifikation der gesamten Verarbeitungskette (Z3, F3) und stellt sicher, dass Sender und Empfänger funktional wie vorgesehen arbeiten (F1, F6). Durch seine flexible Parametrierbarkeit (Z5) sowie die Möglichkeit zur Variantenanalyse verschiedener FFT-Konfigurationen (NF6) unterstützt es gezielt die Bewertung von Durchsatz und Latenz im Sinne der Anforderung NF2. Darüber hinaus dokumentiert es den methodischen Ablauf der Implementierung und schafft damit die Grundlage für eine systematische Integration der High-Level-Synthese gemäß Z7 und NF4.

Zusammenfassung für die weitere Umsetzung:

Für Implementierung und Integration gelten: Abtastrate $f_s = 100$ kHz, $CP = 1/8$, FFT-Längen $N \in \{64, 128, 256\}$, 16-QAM als Arbeitsmodus, Comb-Piloten mit Abstand 7 (nur zur Standardnähe, ohne Nutzung im AWGN) und Guardbänder an beiden Bandenden. Diese Parametrierung ermöglicht einen fairen Variantenvergleich, hält die Bruttoreate über N_{FFT} konstant und bereitet die HLS-Umsetzung mit klaren Schnittstellen, Puffergrößen und Wortlängen vor.

4.3 Zielarchitektur des OFDM-Systems

4.3.1 Blockdiagramm und Partition

Ziel der Architekturkonzeption ist es, eine effiziente Aufteilung der OFDM-Verarbeitungskette zwischen DSP und FPGA zu definieren, um die Rechenlast optimal zu verteilen.

Bereits in der Konzeptphase wurde erkannt, dass der FPGA aufgrund seiner hohen Parallelisierbarkeit besonders geeignet ist, rechenintensive Signalverarbeitungsblöcke wie die FFT und IFFT zu übernehmen[26]. Diese Operationen weisen eine hohe Rechenkomplexität und einen deterministischen Ablauf auf, was sie ideal für eine hardwarebasierte Implementierung macht [27].

Der DSP hingegen eignet sich aufgrund seiner Flexibilität und Programmierbarkeit vor allem für sequentielle, steuerungorientierte Aufgaben wie Symbol-Mapping, Modulation (z. B. 16-QAM), Steuerlogik und Ablaufkontrolle. In der Architekturplanung wurde jedoch von Beginn an vorgesehen, verschiedene Partitionierungsvarianten zu analysieren, um den optimalen Hardware-Software-Split zu identifizieren. Dabei wurden drei mögliche Szenarien konzipiert:

1. **"DSP-only"**: Sämtliche Verarbeitungsschritte der OFDM-Kette – einschließlich Mapping, Modulation, Bildung des Spektralarrays, Datensubträger extrahieren, IFFT/FFT und Steuerlogik – werden vollständig auf dem DSP ausgeführt. Dieser Fall dient als Referenz, um den theoretischen und praktischen Rechenaufwand auf einer rein softwarebasierten Implementierung zu bewerten.
2. **"Geteilte Verarbeitung (DSP/FPGA)"**: Der DSP übernimmt ausschließlich Steuerlogik, Buskommunikation (über EMIFA) und Ablaufkontrolle, während der FPGA sämtliche signalverarbeitenden Aufgaben ausführt, d. h. Symbol-Mapping, Modulation (16-QAM), Bildung des Spektralarrays, Datensubträger extrahieren, Hinzufügen des Cyclic Prefix sowie IFFT/FFT-Berechnung. Dieses Szenario in Abbildung 4.6 bildet das eigentliche Hardware-Software-Codesign ab und sollte den Nutzen der Funktionsauslagerung – insbesondere hinsichtlich Rechenzeit, Parallelität und Durchsatz – aufzeigen.

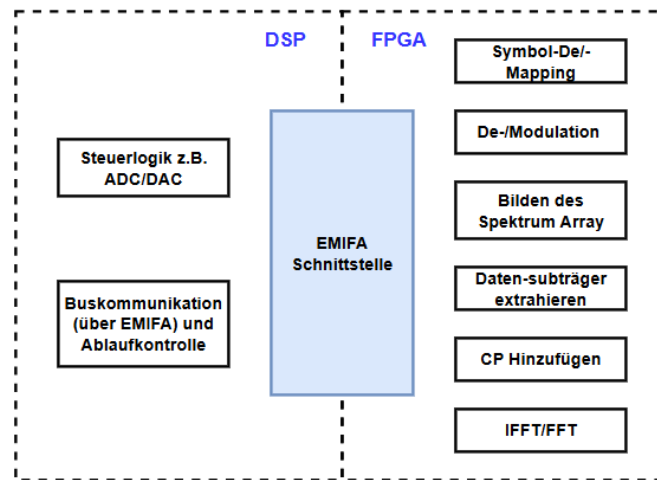


Abbildung 4.6: Konzeptionelle Partitionierungsvarianten des OFDM-Systems (Geteilte Verarbeitung)

3. **"Teilweise Auslagerung"**: Der DSP führt Steuerlogik, Buskommunikation (über EMIFA), Ablaufkontrolle, Symbol-Mapping, Modulation und Cyclic Prefix Hinzufügen weiterhin selbst aus, während die Bildung des Spektralarrays, das Extrahieren der Datensubträger und die IFFT/FFT auf den FPGA ausgelagert werden (siehe Abbildung 4.7). Dieses Szenario wurde ursprünglich als Zwischenlösung betrachtet, um die Auswirkungen einer teilweisen Hardwarebeschleunigung zu untersuchen.

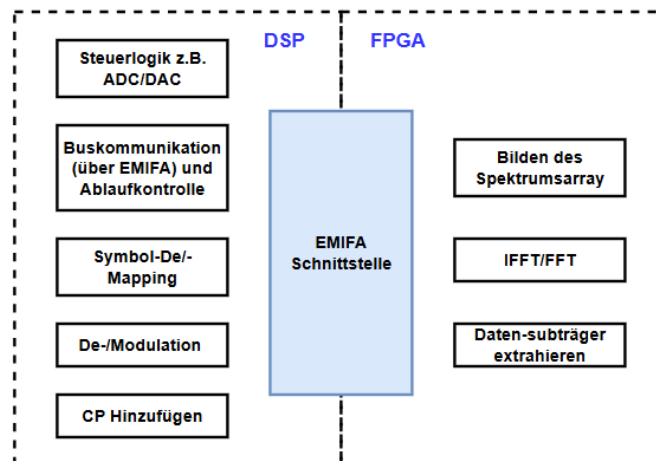


Abbildung 4.7: Konzeptionelle Partitionierungsvarianten des OFDM-Systems (Teilweise Auslagerung)

Bereits in der Konzeptphase zeigt sich, dass insbesondere Fall 2 die effizienteste Aufteilung darstellt. Denn die vollständige Auslagerung der Signalverarbeitungskette auf den FPGA reduziert nicht nur die DSP-Last deutlich, sondern minimiert auch die Anzahl der notwendigen EMIFA-Zugriffe. Im Gegensatz dazu führt die hybride Variante (Fall 3) zu einer erhöhten Kommunikationslatenz, da mehrere EMIFA-Buszyklen pro OFDM-Frame erforderlich sind, um Zwischenresultate auszutauschen. Die dadurch entstehenden Buszugriffszeiten kompensieren den möglichen Rechenzeitgewinn teilweise. Für die FFT-Länge $N = 128$ ergibt sich ein deutlicher Unterschied:

- **Im Fall 2 "Geteilte Verarbeitung (DSP/FPGA)"** reduziert sich die Anzahl der Buszugriffe auf lediglich 12 Schreib- und 128 Leseoperationen, wodurch die effektive Buszeit auf rund $8.4 \mu\text{s}$ sinkt.
- **Im Fall 3 "Teilweise Auslagerung"** sind insgesamt 128 Schreib- und 128 Lesezugriffe erforderlich, was bei einer Buszugriffszeit von 60 ns eine reine Datentransferzeit von etwa $15.36 \mu\text{s}$ ergibt.

Diese Analyse zeigt, dass bei Fall 3 die Datenübertragung über die EMIFA-Schnittstelle zum Engpass wird, wodurch mögliche Vorteile der Hardwarebeschleunigung wieder verloren gehen, während die vollständige Funktionsverlagerung (Fall 2) eine deutlich effizientere Gesamtausführung ermöglicht. Die konzipierte Zielarchitektur basiert somit auf dem Prinzip, rechenintensive und stark parallelisierbare Verarbeitungsschritte auf den FPGA auszulagern und den DSP als steuernde und koordinierende Instanz einzusetzen. Diese Partitionierung bildet die Grundlage für die in den folgenden Abschnitten beschriebene Kommunikationsarchitektur und Datenflussplanung. Die gewählte DSP/FPGA-Partition adressiert NF1 und Z4, begründet den Ressourcen-/Performance-Ausgleich sowie die Durchsatz-/Latenzziele nach NF2, reduziert EMIFA-Transfers im Sinn von Z1 und F2 und sichert durch Messvorkehrungen die Nachvollziehbarkeit gemäß NF6. Zur späteren Bewertung der Partitionierung wurden bereits in der Konzeptphase Messmethoden vorgesehen: Der DSP sollte Taktzyklusinformationen über die Profiling-Funktion der Entwicklungsumgebung erfassen, während im FPGA interne Taktzähler zwischen den Zuständen *start* und *done* integriert werden sollten. Damit ist sichergestellt, dass die spätere Implementierung quantitative Vergleiche zwischen beiden Plattformen erlaubt, ohne die Architektur selbst zu verändern.

4.4 Kommunikationsarchitektur DSP-FPGA über EMIFA

In diesem Abschnitt werden die Konzepte zur Datenübertragung zwischen DSP und FPGA vorgestellt, die auf dem External Memory Interface A (EMIFA) basieren. Der EMIFA dient dabei als gemeinsame physikalische Verbindung für Steuer- und Datensignale und bildet einen zentralen Bestandteil der Systemarchitektur. Die Auswahl und Ausgestaltung des Kommunikationskonzepts hat große Einfluss auf die Effizienz der Gesamtarchitektur, insbesondere im Hinblick auf das geplante Hardware-Software-Codesign. Die hier beschriebenen Entwurfsentscheidungen stellen somit eine wesentliche Grundlage für die spätere Implementierung dar.

4.4.1 Datenübertragung über EMIFA: Wort- vs. Blockweise

Die Kommunikation zwischen DSP und FPGA erfolgt über das EMIFA, das als parallel arbeitende Busarchitektur ausgelegt ist [3] [4]. Im Rahmen dieser Arbeit wurden zwei Varianten für die Datenübertragung evaluiert, die sich hinsichtlich Busbelastung, Steuerlogik und Effizienz deutlich unterscheiden.

Die erste Variante sieht eine direkte, wortweise Übertragung über den EMIFA-Bus vor. Dabei werden die Lese- und Schreibzugriffe vom DSP aus direkt gestartet, wodurch der Prozessor aktiv in jede Datenbewegung eingebunden ist. Wie bereits in Kapitel 2.5 erläutert, führt dieses Verfahren jedoch zu einer starken Bindung des DSP an die Buszugriffe und limitiert damit die parallele Ausführung von Rechenoperationen, insbesondere bei hoher Datenrate.

Die zweite, bevorzugte Variante nutzt den auf der Plattform verfügbaren Enhanced Direct Memory Access (EDMA) Controller, um Daten blockweise über den EMIFA zu übertragen [19] [30]. Der EDMA übernimmt dabei die eigenständige Durchführung der Transfers, sodass der DSP entlastet und für andere Aufgaben freigehalten wird. Diese Architektur entspricht dem Ziel eines effizienten Hardware-Software-Codesigns, da sie eine klarere Trennung von Steuer- und Verarbeitungspfaden ermöglicht und die parallele Ausführung von Kommunikations- und Rechenoperationen unterstützt. Eine Beschreibung der grundlegenden Eigenschaften des EDMA findet sich in Abschnitt 2.5.3.

Im Projektverlauf wurden beide Übertragungsarten konzeptionell betrachtet und vorbereitend untersucht, da sie jeweils unterschiedliche Anforderungen an Steuerlogik und

Systemauslastung stellen. Für das finale System wurde die EDMA-basierte Übertragung als bevorzugte Lösung ausgewählt. Ausschlaggebend hierfür ist die Möglichkeit, große Datenmengen effizient und mit minimaler DSP-Core Belastung zu übertragen. Insbesondere im Hinblick auf zukünftige Erweiterungen – beispielsweise die Integration von signalverarbeitenden Komponenten auf dem DSP – bietet der EDMA-Ansatz klare Vorteile hinsichtlich Datendurchsatz und deterministischem Timing. Durch die Nutzung des EDMA-Controllers können Transfers vollständig autonom und parallel zur restlichen DSP-Verarbeitung ablaufen. Dies ermöglicht eine höhere Busauslastung, reduziert Interrupt- und Polling-Overhead und entlastet den DSP-Core signifikant. Trotz des höheren initialen Konfigurationsaufwands stellt die EDMA-basierte Übertragung langfristig die robustere und skalierbarere Lösung für ein leistungsfähiges DSP-FPGA-System dar.

4.4.2 EMIFA-Empfangsmodul auf FPGA-Seite

Zur Herstellung einer bidirektionalen Verbindung zwischen DSP und FPGA wurde auch auf FPGA-Seite ein spezielles EMIFA-Empfangsmodul entwickelt. Dieses Modul bildet die Gegenstelle zur EMIFA-Schnittstelle auf dem DSP und ermöglicht die kontrollierte Aufnahme sowie Weiterleitung der Datenströme an nachgelagerte Verarbeitungseinheiten, wie in der Abbildung 2.16 Beschrieben ist.

Ein zentrales Ziel des Entwurfs ist es, die im Kapitel 2.5 beschriebene maximale Datenrate gemäß Gleichung 2.21 möglichst vollständig zu erzielen. Dazu werden alle Adress-, Daten- und Steuersignale aus dem EMIFA-Bus auf FPGA-Seite in einer zentralen Empfangslogik verarbeitet. Die interne Taktung und Synchronisierung des Moduls wird in nächsten Abschnitt 4.4.3 näher erläutert.

Das Empfangsmodul wurde außerdem so ausgelegt, dass zukünftige Erweiterungen der Verarbeitungskette auf HLS-Ebene, beispielsweise durch eine optionale Vorwärtsfehlerkorrektur (z. B. Faltungscodes mit Viterbi-Decoder [59]), ohne strukturelle Anpassungen an der Schnittstelle möglich sind. Erweiterte Datenpfade oder zusätzliche Steuerinformationen können über den bestehenden Adressbereich eingebunden werden, was eine einfache Erweiterung und Wiederverwendbarkeit sicherstellt.

Durch seine modulare Struktur stellt das EMIFA-Empfangsmodul eine robuste und flexible Basis für die Anbindung beliebiger HLS-basierter Funktionsblöcke im FPGA dar.

4.4.3 EMIFA-Interface-Taktung: asynchron vs. synchron

Für die Busanbindung zwischen DSP und FPGA stehen grundsätzlich zwei Taktkonzepte zur Verfügung: ein vollständig asynchroner Betrieb mit getrennten Taktdomänen sowie ein synchrones Modell, bei dem der vom DSP generierte EMIFA-Takt wie im Abschnitt 2.5.1 beschrieben ist, direkt im FPGA verwendet wird. Beide Varianten wurden in der Konzeptphase hinsichtlich ihrer Machbarkeit, Komplexität und Leistungsfähigkeit untersucht.

Im asynchronen Ansatz betreibt der DSP den EMIFA mit 100 MHz, während die interne Logik im FPGA mit 50 MHz arbeitet. Zur sicheren Domänenüberquerung sind zwei-stufige Synchronisierer sowie asynchrone FIFO-Puffer erforderlich, um Metastabilität zu vermeiden [51]. Diese zusätzliche Synchronisation erhöht die Buszugriffszeiten auf etwa 130 ns pro Transfer. Bei einer Busbreite von 16 Bit und $t_{eff} \approx t_{EMIFA_Zyklus}$ ergibt sich gemäß Gleichung 2.21 ein maximaler Datendurchsatz von:

$$D = \frac{16 \text{ Bit}}{130 \text{ ns}} = 123.1 \text{ Mbit/s.}$$

Zwar bietet dieser Aufbau eine hohe Robustheit gegenüber Taktdifferenzen, limitiert jedoch die Übertragungsleistung deutlich.

Das synchrone Konzept nutzt den EMIFA-Takt des DSP direkt als Referenzsignal für die Buslogik im FPGA. Dadurch entfallen zusätzliche Synchronisationszyklen, und die Steuersignale (CS, WEN, OEN) lassen sich ohne Verzögerung auswerten. Die effektive Buszugriffszeit reduziert sich auf rund $t_{eff} \approx 60 \text{ ns}$, was einen theoretischen Durchsatz von

$$D = \frac{16 \text{ Bit}}{60 \text{ ns}} = 266.7 \text{ Mbit/s}$$

ermöglicht. Der höhere Implementierungsaufwand im synchronen Betrieb ergibt sich aus der Notwendigkeit eines präzisen FPGA-internen Timing-Designs.

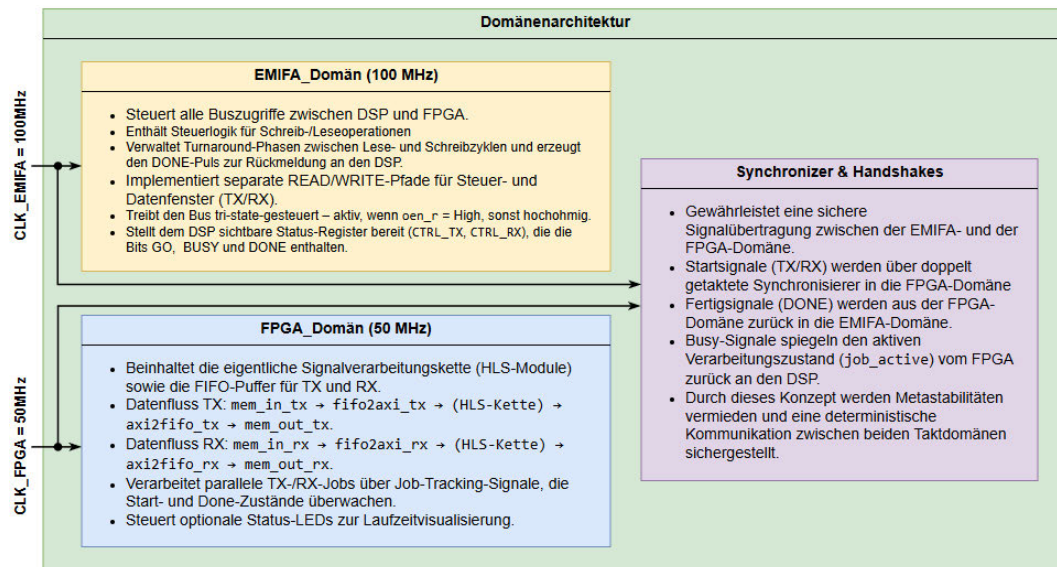


Abbildung 4.8: EMIFA-Interface-Domänenkonzepte (synchron)

Abbildung 4.8 veranschaulicht den Aufbau der Schnittstelle im synchronen Betrieb. Die EMIFA-Domäne (gelb) steuert alle Buszugriffe, während in der FPGA-Domäne (blau) die eigentliche Signalverarbeitung stattfindet. Der Übergang erfolgt über eine entkoppelnde Synchronisationsebene (violett), die mittels FIFO-Puffer und zweistufiger Handshake-Logik eine sichere Übergabe der Daten gewährleistet.

Um die Vorteile beider Konzepte zu kombinieren, wurde ein hybrider Taktarchitekturansatz umgesetzt:

- Die Buslogik des EMIFA-Interfaces arbeitet synchron zum vom DSP gelieferten Takt, was den maximalen Datendurchsatz sichert.
- Die interne Verarbeitung im FPGA erfolgt asynchron zur EMIFA-Domäne und wird über FIFO-Strukturen entkoppelt.

Diese Architektur erfüllt zentrale Anforderungen aus Z1 (EMIFA-Datenanbindung), F2 (EMIFA-Datenübertragung) sowie den nicht-funktionalen Zielen NF1 (zeitgerechte Verarbeitung) und NF2 (Durchsatzoptimierung). Sie bildet zugleich die Grundlage für die in Abschnitt 4.4.4 vorgestellte Integration der internen Datenpfade und Pufferarchitektur.

4.4.4 Datenpfade und Pufferkonzept

Aufbauend auf der in Abschnitt 4.4.3 erläuterten hybriden Taktarchitektur wurde ein Pufferkonzept entwickelt, das die zuverlässige Übergabe von Daten zwischen dem synchron getakteten EMIFA-Interface und der asynchronen, intern getakteten Signalverarbeitung im FPGA sicherstellt. Abbildung 4.9 zeigt den strukturellen Aufbau der entkoppelnden Pufferstufen. Während das EMIFA-Interface auf FPGA-Seite synchron zum vom DSP gelieferten Bustakt betrieben wird, erfolgt die interne Verarbeitung mit einer abweichenden Taktfrequenz. Um diesen Taktdomänenübergang sicher zu gestalten, kommen FIFO-Strukturen zum Einsatz, die sowohl eine entkoppelte Kommunikation als auch eine Zwischenspeicherung der übertragenen Daten ermöglichen [30].

Die Pufferarchitektur folgt dabei der Datenrichtung und sieht für beide Kommunikationsrichtungen – vom DSP in das FPGA (Senderpfad) und zurück (Empfängerpfad) – entsprechende FIFO-Elemente vor. Diese übernehmen die saubere Trennung zwischen Kommunikations- und Verarbeitungslogik und gewährleisten stabile Systemzustände auch bei schwankender Last oder Taktung.

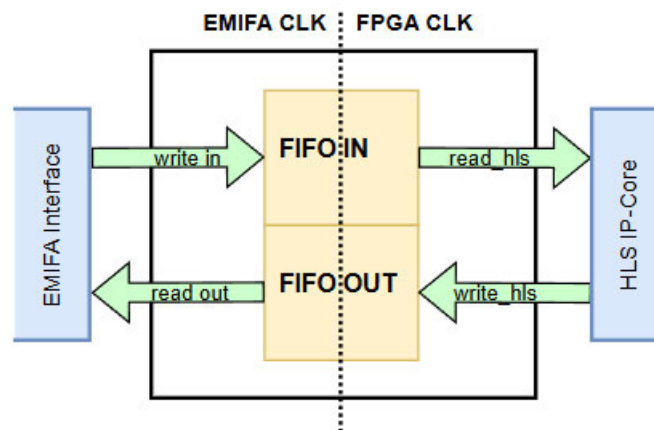


Abbildung 4.9: Pufferkonzept des DSP-FPGA-Systems

Die Pufferstrategie ermöglicht eine präzise Steuerung des Datenflusses und bildet eine flexible Schnittstelle, die bei Bedarf um zusätzliche Verarbeitungsstufen erweitert werden kann – beispielsweise für zukünftige Systemfunktionen wie Vorwärtsfehlerkorrektur oder zusätzliche Modulationsverfahren. Eine enge Verzahnung mit der Steuerarchitektur über die in Abschnitt 4.4.5 beschriebene Register-Map ist dabei vorgesehen.

Der gewählte Ansatz erfüllt zentrale Anforderungen an Busstabilität (Z1, F2), zeitgerechte Verarbeitung (NF1), effizienten Ressourceneinsatz (NF2) und Erweiterbarkeit (F8). Die entwickelte Architektur bildet damit die konzeptionelle Grundlage für die nachfolgende Anbindung der HLS-basierten Module über eine AXI4-Stream-Schnittstelle (vgl. Abschnitt 4.5).

4.4.5 Register-Map statt GPIO-Steuerung

Für die Steuerung und Statusabfrage zwischen DSP und FPGA wurde im Konzept eine Register-Map-basierte Kommunikation vorgesehen, anstelle der Verwendung einzelner GPIO-Leitungen. Ziel dieser Entscheidung war es, eine skalierbare und strukturierte Schnittstelle zu schaffen, die bei minimalem Hardwareaufwand eine flexible Steuerung ermöglicht [21] [48]. Zwar wäre auch eine signalbasierte GPIO-Steuerung prinzipiell umsetzbar gewesen, sie hätte jedoch erhebliche Einschränkungen mit sich gebracht: Die Anzahl der verfügbaren GPIO-Leitungen auf der Plattform ist begrenzt, und mit wachsender Systemkomplexität wäre eine signalbasierte Steuerung schnell an ihre Grenzen gestoßen. Zudem hätte die Kodierung von Steuerinformationen über parallele Signale zu erhöhter Komplexität, erhöhter Fehleranfälligkeit und eingeschränkter Erweiterbarkeit geführt [60].

Das gewählte Register-Map-Konzept löst diese Probleme, indem sämtliche Steuer- und Statusinformationen über definierte Adressbereiche im EMIFA-Raum zugänglich gemacht werden. Der DSP kann dabei auf FPGA-seitige Register zugreifen, als handle es sich um externen Speicher, während der FPGA intern die eingehenden Schreib- und Lesezugriffe auswertet.

Vorgesehen sind dabei zwei logische Registerbereiche:

- **Control-Register (CTL):** Steuerbits zur Auslösung von Verarbeitungszyklen, z. B. `Start_TX`, `Start_RX`.
- **Status- und Timing-Register (STAT/TIME):** Rückmeldungen wie `BUSY`, `DONE` oder ein optionaler Taktzähler zur Zeitmessung.

Die Kommunikation zwischen DSP und FPGA reduziert sich somit auf einfache Lese- und Schreibzugriffe über den EMIFA-Bus, was die Steuerlogik klar strukturiert und den

Integrationsaufwand erheblich minimiert. Die Speicherabbildung bietet eine deterministische, adressbasierte Kontrolle, die sich bei Bedarf um zusätzliche Register problemlos erweitern lässt.

Gegenüber einer GPIO-basierten Lösung ergeben sich mehrere Vorteile:

- **Erweiterbarkeit:** Neue Funktionen können durch zusätzliche Registeradressen ergänzt werden, ohne Hardwareänderungen.
- **Übersichtlichkeit:** Alle Steuerinformationen sind zentral im Adressraum organisiert.
- **Integrationsfreundlichkeit:** Die Registeransteuerung ist direkt mit der EMIFA-Architektur kompatibel.

Das Register-Map-Design ist bewusst minimalistisch, aber funktional gehalten. Bereits mit zwei Registern lässt sich der Datenfluss vollständig steuern und überwachen. Gleichzeitig bleibt die Architektur flexibel genug, um zukünftige Erweiterungen auf System- oder Funktionsebene aufzunehmen. Die gewählte Lösung unterstützt damit die Kommunikationsanforderungen aus F2, die Modularitätsziele aus NF6 sowie die Echtzeit- und Skalierbarkeitskriterien gemäß NF1 und NF2.

4.5 Datenpfad-Integration: FIFO $\leftrightarrow$ AXI4-Stream

Ein weiteres zentrales Ziel der Architekturplanung bestand darin, die über EMIFA bereitgestellten Daten nahtlos in die FPGA-internen HLS-Module zu überführen. Zu diesem Zweck wurde ein verbindendes Schnittstellenkonzept entwickelt, das FIFO-Strukturen im EMIFA-Bereich mit der im FPGA etablierten AXI4-Stream-Kommunikation kombiniert (siehe Abbildung 4.10) [13].

Die Wahl der AXI4-Stream-Schnittstelle erfolgte aufgrund ihrer weiten Verbreitung, klar definierten Signalstruktur und hohen Kompatibilität mit gängigen HLS-Werkzeugen wie AMD Vivado HLS oder Vitis HLS [47]. Durch die native Unterstützung in Synthese- und Simulationsumgebungen kann der Datenfluss zwischen IP-Blöcken frühzeitig getestet und effizient integriert werden. Darüber hinaus unterstützt das AXI4-Stream-Protokoll kontinuierliche Datenströme und optionale Burst-Übertragungen – auch wenn diese im EMIFA-Kontext nicht genutzt werden, bietet die interne Stream-Verarbeitung klare Vorteile in Bezug auf Modularität und Durchsatz [28].

Für die Übergabe zwischen dem EMIFA-FIFO-System und der AXI4-Stream-Welt wurde ein zweistufiges Integrationsprinzip entworfen:

- **fifo2axi:** Dieses Modul liest Daten aus den FIFO-Puffern der EMIFA-Domäne und übergibt sie als AXI4-Stream an die nachfolgenden HLS-Module.
- **axi2fifo:** Hier werden die verarbeiteten Daten aus dem AXI4-Stream der HLS-Kette zurück in FIFO-Strukturen geschrieben, sodass sie über den EMIFA-Bus wieder an den DSP übertragen werden können.

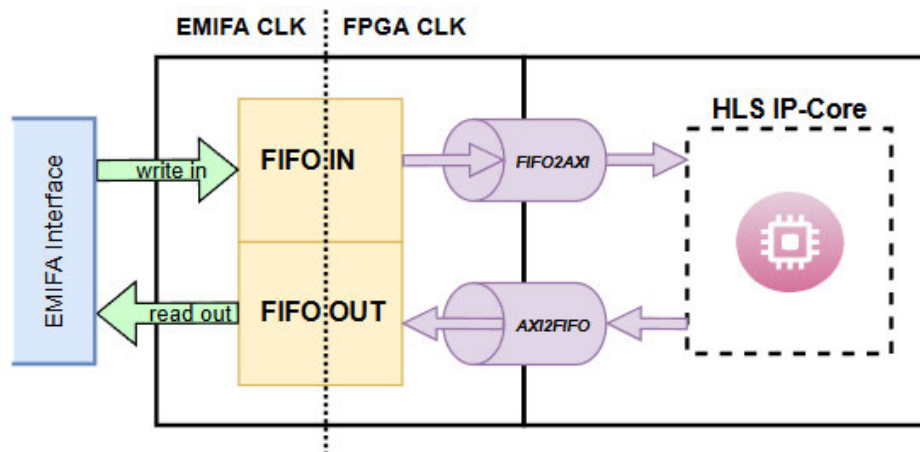


Abbildung 4.10: Datenpfad-Integrationskonzept FIFO ↔ AXI4-Stream

Beide Brückenmodule nutzen die Standard-Signale `tdata`, `tvalid`, `tready` und `tlast`, die durch das AXI4-Stream-Protokoll definiert sind. Dadurch wird eine klare Trennung zwischen blockweiser, speicherorientierter Datenbereitstellung (EMIFA) und paketbasierter, kontinuierlicher Datenverarbeitung (AXI4-Stream) erreicht. Die Schnittstelle fungiert somit als funktionale Brücke und erlaubt eine flexible, performante Integration der HLS-basierten OFDM-Verarbeitung.

Die gewählte Architektur stärkt das modulare Systemdesign (NF1, NF2), unterstützt die effiziente HLS-Implementierung zentraler Funktionsblöcke (Z7, F1, F6) und erfüllt die methodischen Anforderungen an eine dokumentierbare und wiederverwendbare Entwicklung gemäß NF4.

4.6 HLS-IP-Kette: Auswahl und Architekturprinzipien

In diesem Abschnitt wird die konzeptionelle Planung der HLS-basierten Verarbeitungsmodule beschrieben. Ziel ist es, eine modulare, effizient synthesefähige IP-Kette aufzubauen, die zentrale Funktionen der OFDM-Signalverarbeitung übernimmt und sich flexibel in die AXI4-Stream-Verarbeitungspipeline integrieren lässt. Der Fokus liegt auf Auswahl, Strukturierung und Optimierung der HLS-Module hinsichtlich Datenfluss, Parallelisierbarkeit und Schnittstellenkompatibilität.

Der Entwicklungsansatz erfolgt schrittweise: Zunächst wird ein erstes funktionales Inkrement realisiert, das wesentliche Module des OFDM-Sendepfads umfasst. Aufbauend darauf werden Architekturprinzipien identifiziert, die für eine performante, skalierbare und wiederverwendbare HLS-Entwicklung zentral sind.

4.6.1 Funktionsumfang (erstes Inkrement)

Im ersten Entwicklungsschritt wurde ein begrenzter, aber funktionsfähiger Teil der OFDM-Verarbeitungskette als Proof-of-Concept in HLS realisiert. Grundlage für die Auswahl der umzusetzenden Module war das zuvor entwickelte MATLAB-Simulationsmodell (vgl. Abschnitt 4.2), das die gesamte Sende- und Empfangskette modelliert und zur Referenz für Struktur und Funktionsweise der Hardware-Module diente.

Ziel ist es, auf Basis dieses Modells die prinzipielle Machbarkeit der High-Level-Synthese für zentrale Verarbeitungsschritte zu evaluieren und ein modulares Fundament für die spätere Systemerweiterung zu schaffen. Das erste Inkrement konzentrierte sich auf den Sendepfad und umfasste folgende Kernfunktionen:

- Symbol-Mapping für 16-QAM,
- Modulation und Arraybildung für den IFFT-Eingang,
- Implementierung der IFFT auf Basis eines Radix-2-Algorithmus [24],
- Hinzufügen des Cyclic Prefix.

Diese Auswahl wurde getroffen, weil die genannten Verarbeitungsschritte im MATLAB-Modell bereits funktional abgebildet waren und sich gleichzeitig gut als modulare HLS-Blöcke mit klaren Schnittstellen realisieren lassen. Darüber hinaus stellen sie arithmetisch

anspruchsvolle und parallelisierbare Aufgaben dar, was eine gezielte Untersuchung von Latenz, Ressourcenauslastung und Syntheseeffizienz im Vergleich zur DSP-Referenz ermöglichte [36].

Für die Implementierung der IFFT wurde der Radix-2-Algorithmus gewählt. Eine FFT der Länge N erfordert in dieser Struktur:

$$\text{komplexe Multiplikationen:} \quad \frac{N}{2} \cdot \log_2(N) \quad (4.1)$$

$$\text{komplexe Additionen:} \quad N \cdot \log_2(N) \quad (4.2)$$

Zum Vergleich: Bei einer Radix-4-Implementierung reduziert sich die Anzahl der Multiplikationen auf:

$$\frac{3N}{8} \cdot \log_2(N) \quad (4.3)$$

Die Anzahl der Additionen bleibt unverändert. Damit ergibt sich eine potenzielle Rechenzeiterparnis von etwa 25 % [6].

Trotz dieses Vorteils fiel die Wahl auf Radix-2 – insbesondere aus folgenden Gründen:

- **FFT-Längenabhängigkeit:** Die in diesem Projekt gewählte FFT-Länge $N = 128$ ist keine Potenz von 4^k und somit für Radix-4 ungeeignet.
- **Wiederverwendbarkeit:** Das Radix-2-Design ist generisch einsetzbar und lässt sich einfacher auf andere FFT-Längen anpassen.

Die Module wurden gemäß den Optimierungsprinzipien **DATAFLOW** [8], **PIPELINE** [61] und **LOOP UNROLLING** [62] entworfen, um eine möglichst hohe Parallelität zu erreichen. Wie in Abschnitt 2.3.3 beschrieben, ermöglichen diese Maßnahmen eine kontinuierliche Abarbeitung entlang der Verarbeitungstrecke ohne Blockierung zwischen den Modulen.

Das erste HLS-Inkrement diente damit vor allem der Evaluierung des Entwurfsflusses, der Schnittstellenintegration und der Synthesecharakteristik. Die realisierten Module decken kritische Rechenpfade ab und bilden eine belastbare Grundlage für die schrittweise Erweiterung zur vollständigen OFDM-Kette (vgl. Abbildung 2.6 und 2.9).

4.6.2 HLS-Entwurfsprinzipien zur Taktoptimierung

Wie bereits im Grundlagenkapitel beschrieben, hängt die erreichbare Taktfrequenz eines mittels High-Level-Synthese generierten IP-Cores nicht ausschließlich von der Zielplattform ab, sondern wesentlich von der Struktur des zugrunde liegenden Algorithmus, der Datenflussorganisation und der Anwendung konkreter Optimierungsmaßnahmen im HLS-Entwurfsprozess.

Bereits in der Konzeptphase wurde daher ein besonderer Fokus auf strukturelle Entwurfsentscheidungen gelegt, die eine spätere Taktoptimierung ermöglichen und die erreichbare Frequenz nicht unnötig einschränken. Ziel war es, für alle relevanten Module eine Architektur zu entwerfen, die prinzipiell Taktfrequenzen oberhalb der systemseitigen 50 MHz erlaubt, um sowohl für aktuelle Plattformen als auch für zukünftige Weiterentwicklungen mit höheren Anforderungen gerüstet zu sein.

Im Vordergrund standen dabei keine konkreten HLS-Pragmas, sondern generelle Entwurfsprinzipien, die Synthese und Timing-Verhalten positiv beeinflussen. Die folgenden Leitlinien wurden im Rahmen der Architekturplanung definiert:

1. **Modularisierung und Timing-Entkopplung:** Jeder Hauptfunktionsblock (z. B. Mapping, Modulation, IFFT) wurde als separates Modul geplant, um kritische Pfade lokal zu halten. Dies erlaubt gezielte Taktoptimierungen innerhalb eines Moduls, ohne globale Instabilitäten zu riskieren.
2. **Abhängigkeitsarme Schleifenstruktur:** Schleifen wurden bewusst so formuliert, dass inter-iterative Abhängigkeiten minimiert sind. Dies begünstigt spätere Anwendung von *loop unrolling* oder *loop flattening* und ermöglicht geringe *initiation intervals* in pipelinisierten Schleifen [62].
3. **Dataflow-Orientierung zwischen Modulen:** Die Architektur wurde auf eine dataflow-fähige Verknüpfung zwischen Modulen hin geplant. Dadurch können mehrere Blöcke parallel arbeiten, was sequentielle Flaschenhälse vermeidet und die globale Taktbegrenzung reduziert [8].

4. **Ressourcen- und Speicherbewusste Gestaltung:** Um Speicherzugriffe takt-effizient zu gestalten, wurden Arrays und Zwischenspeicher so ausgelegt, dass Partitionierung und parallele Zugriffe möglich sind. Dies erlaubt sowohl Ressourcenoptimierung als auch parallele Speicherzugriffe [61].

Ziel dieser Prinzipien war nicht primär eine sofortige Taktmaximierung, sondern das Schaffen robuster struktureller Voraussetzungen, um potenzielle Taktreserven systematisch erschließen zu können. Das Architekturkonzept ermöglicht damit eine spätere Portierung auf höher getaktete Plattformen oder die Integration in Multi-Clock-Systeme.

Die Umsetzung dieser Entwurfsrichtlinien zählt direkt auf mehrere Anforderungen ein: Sie unterstützt die systematische Aufgabenteilung im Codesign (Z4), sichert hohe Durchsatz- und Latenzperformance (NF2), berücksichtigt Aspekte in Bezug auf die Bitbreite (NF3) und dokumentiert einen nachvollziehbaren HLS-Vorgehensweg (NF4).

4.7 Systemintegration und Duplex-Fähigkeit

In der Konzeptphase wurde das System gezielt so ausgelegt, dass es eine vollständige Duplex-Fähigkeit aufweist, d. h. Sende- (TX) und Empfangspfad (RX) können gleichzeitig und unabhängig voneinander betrieben werden. Dieses Ziel wird durch den Einsatz zweier getrennter, parallel arbeitender HLS-IP-Cores realisiert, die physisch nebeneinander im FPGA implementiert sind, (siehe Abbildung 4.11).

Beide IP-Cores verfügen über eigene FIFO-Strukturen und einen vollständig separierten Datenpfad – von der Dateneinspeisung über die Verarbeitung bis hin zur Ergebnisausgabe. Die Architektur gewährleistet damit eine echte Parallelverarbeitung auf Hardwareebene: TX- und RX-Signalwege greifen nicht auf gemeinsame Ressourcen zu und erfordern keine Synchronisation, was insbesondere bei zeitkritischen Kommunikationsanwendungen von Vorteil ist.

Zur Steuerung der Verarbeitungsblöcke wird eine duplexfähige Registerstruktur verwendet. Im EMIFA-Adressraum sind separate Steuer- und Statusregister für beide Pfade vorgesehen, wodurch eine unabhängige Konfiguration und Statusabfrage ermöglicht wird. Der DSP adressiert beide IP-Cores über denselben Bus, ohne dass zusätzliche Steuerleitungen erforderlich wären.

Obwohl der EMIFA-Bus bidirektional ausgelegt ist, unterstützt er keine gleichzeitige Sende- und Empfangsoperation (Vollduplex). Die Architektur trägt dieser Einschränkung Rechnung, indem die parallele Datenverarbeitung intern im FPGA erfolgt, der Datenaustausch mit dem DSP jedoch sequentiell über dedizierte FIFO-Puffer abgewickelt wird. Der DSP kann so kontrolliert und nacheinander auf die Ergebnisse aus TX und RX zugreifen.

Das resultierende Systemkonzept kombiniert hardwareseitige Parallelität mit deterministischer, sequentieller Bussteuerung. Es erlaubt eine skalierbare, klar strukturierte Architektur, bei der die Signalverarbeitung für beide Kommunikationsrichtungen simultan erfolgt, die Busanbindung jedoch kontrolliert sequenziert bleibt.

Die konkrete Signalstruktur, Registerbelegung und physikalische Anbindung der Module werden im folgenden Implementierungskapitel 5.9 ausführlich erläutert. Die vorgesehene Systemarchitektur erfüllt zentrale Anforderungen der Zielplattform: Sie realisiert F1 im Rahmen von Z2, ermöglicht eine vollständige Ende-zu-Ende-Kommunikation gemäß Z6 und stellt durch klar dokumentierte Steuerpfade (NF6) eine konsistente Erweiterbarkeit (F8) sicher.

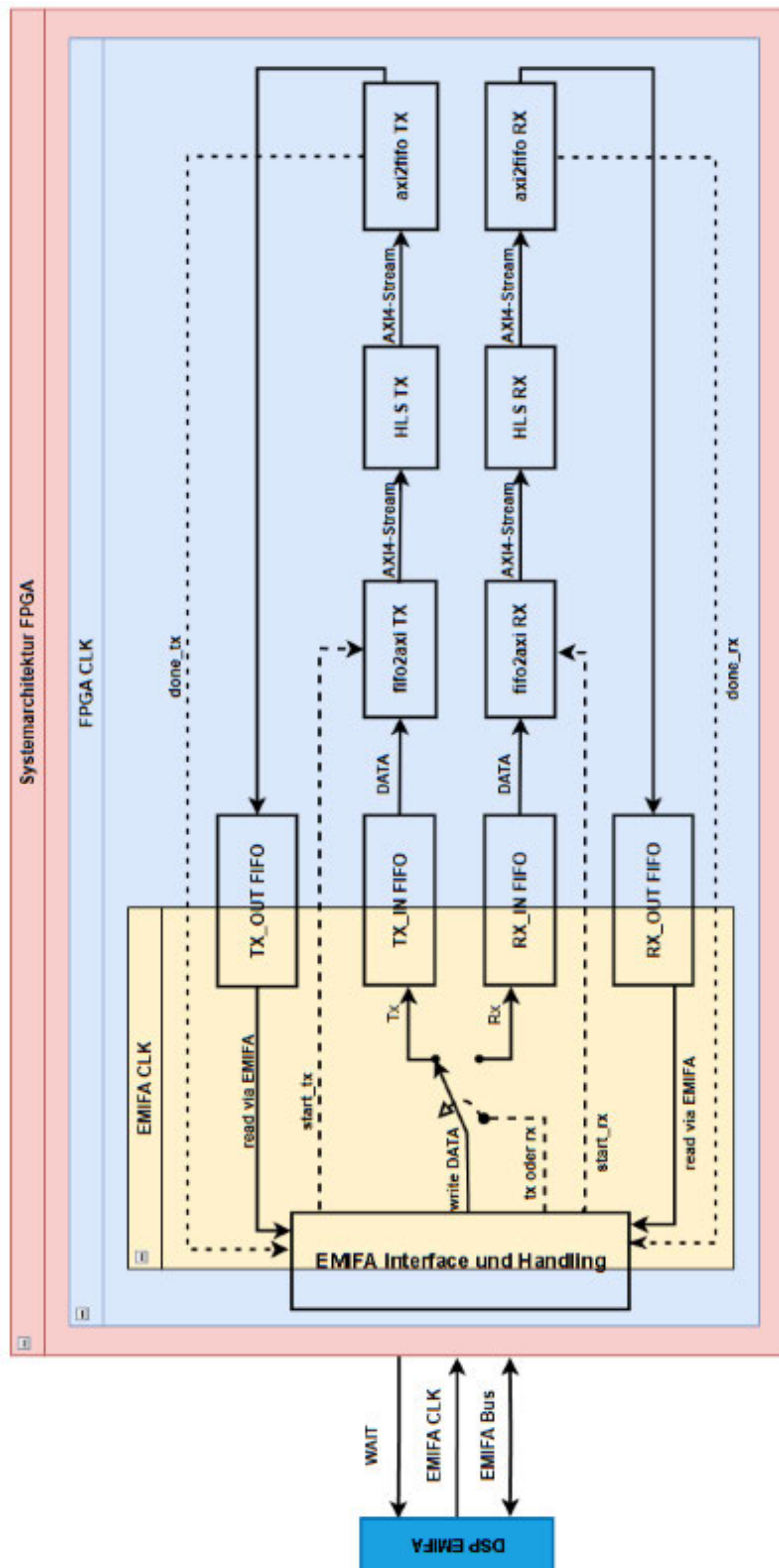


Abbildung 4.11: Systemarchitektur mit getrennten Taktdomänen und Steuerpfaden

5 Implementierung des OFDM-Systems

5.1 Überblick und Zielsetzung

Dieses Kapitel beschreibt die schrittweise Umsetzung des in Kapitel 4 entwickelten OFDM-Systems auf der Zielplattform. Dabei werden die Konfiguration des DSPs, die Implementierung der Kommunikationsschnittstellen auf dem FPGA und die Integration der Verarbeitungskette mittels High-Level-Synthese erläutert.

Ziel ist es, eine duplexfähige und skalierbare Implementierung bereitzustellen, welche die zuvor definierten funktionalen und performanzbezogenen Anforderungen vollständig erfüllt.

5.2 Entwicklungsumgebung

Für die Implementierung des OFDM-Systems wurde eine spezialisierte Entwicklungsumgebung eingerichtet, die sowohl FPGA- als auch DSP-seitige Werkzeuge umfasst. Ziel war es, eine durchgängige und aufeinander abgestimmte Toolchain bereitzustellen, mit der sich sowohl die hardwareseitigen Verarbeitungskomponenten als auch die DSP-gesteuerte Kommunikation und Systemsteuerung effizient entwickeln und testen.

Die Entwicklungsumgebung gliedert sich in drei Hauptbereiche:

- FPGA-Design und Schnittstellenintegration mit AMD Vivado [12],
- HLS-basierte Entwicklung und Simulation von Signalverarbeitungsmodulen mit Vitis HLS [11],
- DSP-Softwareentwicklung und EMIFA-Bussteuerung mit Code Composer Studio [54].

Darüber hinaus wurde MATLAB [42] zur Referenzmodellierung des OFDM-Systems eingesetzt, insbesondere zur Generierung von Testvektoren, zur numerischen Validierung der Signalverarbeitung sowie zur quantitativen Bewertung der HLS-Module.

Die Hard- und Softwarekomponenten wurden über dedizierte JTAG-Schnittstellen programmiert und debuggt: Auf FPGA-Seite kam ein **Xilinx Platform Cable USB II** [10] zum Einsatz, während der DSP über ein **XDS200 Debug Interface** [55] angebunden wurde.

5.2.1 Toolchain, Plattform und Versionen

Die eingesetzten Werkzeuge und Plattformen sind in Tabelle 5.1 zusammengefasst. Bei der Auswahl der Toolversionen wurde darauf geachtet, dass alle Komponenten untereinander kompatibel sind, insbesondere hinsichtlich der IP-Integration, der HLS-Synthese und der Debug-Funktionalität.

Komponente	Tool / Plattform	Version
FPGA-Design (Artix-7)	Vivado (AMD)	2024.2
HLS-Modulentwicklung	Vitis HLS (AMD)	2024.2
DSP-Softwareentwicklung	Code Composer Studio (TI)	12.8.1
Signalverarbeitung / Modell	MATLAB	R2025a
FPGA-Programmierung	Platform Cable USB II (Xilinx)	–
DSP-Debugging	XDS200 USB Debug Probe	–

Tabelle 5.1: Verwendete Tools und Plattformen zur Systementwicklung

5.3 DSP-seitige Implementierung

Die DSP-seitige Umsetzung des OFDM-Systems umfasst die Konfiguration zentraler Hardwareschnittstellen sowie die softwareseitige Ablaufsteuerung der Datenübertragung zwischen DSP und FPGA. Im Fokus stehen dabei das External Memory Interface A (EMIFA) und der Enhanced Direct Memory Access (EDMA), die in Kombination eine effiziente Kommunikation mit dem FPGA ermöglichen. Ergänzend wird die Ansteuerlogik in Form eines Zustandsautomaten beschrieben, welche die Zugriffssequenzen auf die externen Verarbeitungsböcke koordinieren.

5.3.1 EMIFA-Konfiguration

Die Konfiguration des EMIFA-Moduls erfolgte über das Chip-Select-Register CS2, welches Bestandteil der DSP-internen EMIFA-Steuerlogik ist [4] [49]. Die Zuordnung einer externen Peripherie erfolgt über die Festlegung einer Basisadresse im externen Speicherbereich. Dem FPGA wurde die Basisadresse 0x60000000 zugewiesen, wodurch ein direkter, speichergemappter Zugriff auf FPGA-seitige Register und Datenbereiche möglich ist. Die Adressierung erfolgt dabei im DSP über folgenden Zugriffstyp:

```
1 #define FPGA_BASE 0x60000000 /* EMIFA CS2 Basisadresse */
2 #define WRIT_ADDR 0x0001 /* z.B.: Adresse, an dem Daten im FPGA geschrieben werden
   */
3 #define READ_ADDR 0x0002 /* z.B.: Adresse, an dem Daten aus dem FPGA gelesen werden
   */
4
5 typedef struct {
6     int16_t ram[16384]; /* 2^(14) Adressen sind möglich bei 14-Bit Adressbus*/
7 }FPGA_t;
8
9 typedef volatile FPGA_t *FPGA_ovly;
10 static volatile FPGA_t fpga = (FPGA_ovly)FPGA_BASE;
11
12 /* Schreibzugriff */
13 fpga->ram[WRIT_ADDR] = 0x1234;
14
15 /* Lesezugriff */
16 int16_t value = fpga->ram[READ_ADDR];
```

Listing 5.1: Speicherabbildung und Zugriff auf FPGA über EMIFA

Die Buszugriffszeiten wurden anhand des im Konzept vorgesehenen EMIFA-synchronen Datenübertragungsmodells sowie der Anforderungen der FPGA-seitigen Peripherie abgestimmt. Für sowohl Lese- als auch Schreibzugriffe wurden identische Timingparameter konfiguriert:

- $t_{\text{SETUP}} = 10 \text{ ns}$,
- $t_{\text{STROBE}} = 40 \text{ ns}$,
- $t_{\text{HOLD}} = 10 \text{ ns}$.

Zusätzlich wurde ein Turnaround-Zeitintervall (TA) von 10 ns definiert, um bei Richtungswechseln auf dem Bus (Lesen $\leftrightarrow$ Schreiben) eine korrekte Übergabe zu gewährleisten. Das

optionale Handshake-Signal `WAIT` wurde aktiviert und auf FPGA-Seite als high-aktives Steuersignal implementiert [33].

Die korrekte Umsetzung der Konfiguration wurde durch Oszilloskopmessungen an den relevanten EMIFA-Leitungen verifiziert. Abbildung 5.1 zeigt exemplarisch den Signalverlauf während eines Lese- und eines Schreibvorgangs. Die folgenden Steuerleitungen wurden aufgezeichnet:

- CH1: `DSP_CS2N` – Chip Select,
- CH2: `DSP_WEN` – Write Enable,
- CH3: `DSP_OEN` – Output Enable.

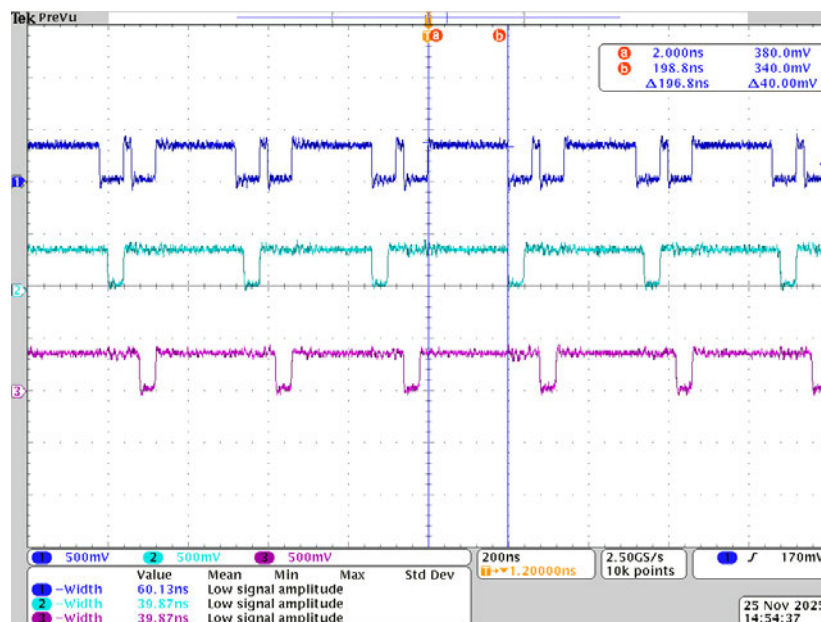


Abbildung 5.1: Signalverlauf während eines Schreib- Lesevorgangs über EMIFA (DSP-Core)

Die aufgezeichneten Signale bestätigen die korrekte Taktung und die Buszugriffszeiten sowie Steuerung entsprechend den Vorgaben des TMS320C6747 Datenblatts [2]. Auffällig ist ein systematisches Pausenverhalten zwischen aufeinanderfolgenden Zugriffen, insbesondere bei Lesezugriffen. Die Leerlaufphasen betragen ca. 200 ns.

Obwohl der EMIFA einen asynchronen Zugriff ohne zusätzliche Wartezyklen unmittelbar an den nächsten anfügen könnte – bis auf die Turnaround-Zyklen [2] – entstehen in

der Praxis dennoch deutlich längere Pausen zwischen zwei Leseoperationen. Dieses Verhalten resultiert jedoch nicht aus der internen EMIFA-Zustandsmaschine, sondern aus der Interaktion zwischen EMIFA und dem DSP-Core. Lesezugriffe („loads“) werden vom DSP-Core stallsynchron behandelt: Der Kern wartet, bis die gelesenen Daten über das EMIFA zurückgeliefert wurden, bevor die nächste Load-Instruktion ausgegeben werden kann. Der Overhead der Instruktionspipeline (Adressupdate, Schleifenverwaltung, Verzweigung) bewirkt daher eine DSP-Core-seitige Latenz, in der der EMIFA keinen neuen Zugriff erhält und der Bus sichtbar inaktiv bleibt. Im Gegensatz dazu werden Schreibzugriffe („stores“) im internen Write-Buffer gepostet und vom EMIFA unabhängig vom Kern abgearbeitet. Der DSP kann dadurch mehrere Stores nahezu ohne zeitlichen Abstand ausgeben.

Die beobachteten unterschiedlichen Pausen zwischen Lese- und Schreiboperationen sind somit nicht auf zusätzliche EMIFA-Timingphasen zurückzuführen, sondern auf die unterschiedliche Behandlung von Loads und Stores im DSP-Core. Der EMIFA selbst erzeugt bei sequenziellen Lesezugriffen keinen zusätzlichen Turnaround, sondern arbeitet jede angeforderte Transaktion unmittelbar ab, sobald der Kern sie anfordert.

Ergänzend zu den exemplarischen Einzeltransaktionen in Abbildung 5.1 enthalten die Oszilloskopmessungen im Anhang (Abbildung A.5 und Abbildung A.6) kontinuierliche Aufzeichnungen für fortlaufende Schreib- und Lesezyklen. Dabei zeigt sich, dass das beschriebene Pausenverhalten bei einer großen Anzahl sequentieller Lesezugriffe besonders deutlich ausgeprägt ist. Die resultierende Busauslastung ist bei Leseoperationen signifikant geringer als bei Schreibvorgängen, wodurch der effektiv erreichbare Datendurchsatz des EMIFA-Interfaces deutlich reduziert wird.

Die beschriebenen Leerlaufphasen begrenzen die Busperformance insbesondere bei datenintensiven Lesezugriffen und stellen damit eine wesentliche architekturbedingte Einschränkung dar. Die Oszilloskopaufnahmen zeigen zudem, dass Schreibzugriffe aufgrund der geringeren internen Zusatzverzögerungen effizienter ablaufen als Leseoperationen, was in der späteren Datenpfadplanung berücksichtigt wurde.

Zur Reduktion der durch die EMIFA-Charakteristik bedingten Einschränkungen wurde im weiteren Verlauf der Einsatz von EDMA-basierten Transfers untersucht. Diese ermöglichen blockweise Zugriffe mit geringerer DSP-Core-Interaktion und höherem effektivem Datendurchsatz.

5.3.2 EDMA-Konfiguration

Für die blockweise Datenübertragung zwischen dem DSP und dem über EMIFA angebundenen FPGA wurde der **EDMA3**-Controller des TMS320C6747 eingesetzt [4]. Zwei DMA-Kanäle wurden so konfiguriert, dass sie den bidirektionalen Datentransfer abdecken: ein Kanal übernimmt Schreibvorgänge (DSP $\rightarrow$ FPGA), der andere Lesetransfers (FPGA $\rightarrow$ DSP).

Wie in Abschnitt 5.3.1 erläutert, ist der FPGA im EMIFA-Adressraum als speichergemappedes Peripheriemodul eingebunden. Dadurch kann der EDMA direkt auf die entsprechenden BRAM- oder Registerbereiche im FPGA zugreifen, ohne dass zusätzliche Adressdekodierungen oder Zwischenschaltungen notwendig sind.

Bei der Initialisierung wurden die erforderlichen Event-Trigger und Transfer-Completion-Interrupts aktiviert. Dies ermöglicht dem DSP eine eindeutige Rückmeldung darüber, ob ein Datenblock erfolgreich geschrieben oder gelesen wurde. Damit lässt sich der weitere Verarbeitungsfluss exakt synchronisieren, ohne auf Polling oder Zeitverzögerungen angewiesen zu sein. Die Transferparameter wurden über die *PaRAM*-Strukturen als eindimensionale Transfers (1D) mit 16-Bit-Datenwörtern konfiguriert. Quelle und Zieladresse inkrementieren linear, die Länge des Transfers ergibt sich aus der Anzahl der 16-Bit-Datenwörter. Jeder Trigger löst die Übertragung eines kompletten Blocks aus, ohne dass eine Intervention des DSP-Core erforderlich ist [33]. Listing 5.2 zeigt eine exemplarische Konfiguration [34].

```
1  #define EDMA_FPGA_WRITE_CH  2
2  #define EDMA_FPGA_READ_CH   3
3  #define EDMA_PARAM_BASE     EDMA3CC_BASE + 0x4000
4  #define EDMA_PARAM_SIZE     32
5
6  typedef volatile struct {
7      uint32_t OPT;
8      uint32_t SRC;
9      uint32_t DST;
10     uint16_t ACNT;
11     uint16_t BCNT;
12     int16_t  SRCBIDX;
13     int16_t  DSTBIDX;
14     uint16_t LINK;
15     uint16_t CCNT;
16 } EDMA_PaPARAM;
17
18 /* PaPARAM für 1D-Transfer mit 16Bit-Elementen AB-sync, ein Trigger */
19 EDMA_PaPARAM *p = (EDMA_PARAM_BASE + EDMA_FPGA_READ_CH * EDMA_PARAM_SIZE);
```

```

20
21 p->OPT      = OPT_TCINTEN | OPT_SYNCDIM_AB | (tcc << OPT_TCC_SHIFT); /* Interrupt
    einschalten */
22 p->SRC       = (uint32)&source_buffer;
23 p->DST       = (uint32)&(fpga->ram[start_index]);
24 p->ACNT      = 2; /* 2 Byte pro Element */
25 p->BCNT      = words; /* Anzahl 16-Bit-Worte */
26 p->SRCBIDX   = 2; /* Nach jedem übertragenen Element, die Source-Adresse um 2
    Byte erhöht */
27 p->DSTBIDX   = 2; /* Nach jedem übertragenen Element, die Destination-Adresse
    um 2 Byte erhöht */
28 p->LINK      = 0xFFFF; /* kein Linking */
29 p->CCNT      = 1; /* 1 Frame*/

```

Listing 5.2: EDMA-Setup für 1D-Blocktransfer zwischen DSP und FPGA

Die korrekte Funktion und das Timing der EDMA-basierten Transfers wurden über Oszilloskopmessungen an den relevanten Steuerleitungen der EMIFA-Schnittstelle überprüft. Abbildung 5.2 zeigt exemplarisch den Signalverlauf bei einem EDMA-gestützten Schreib- und Lesevorgang.

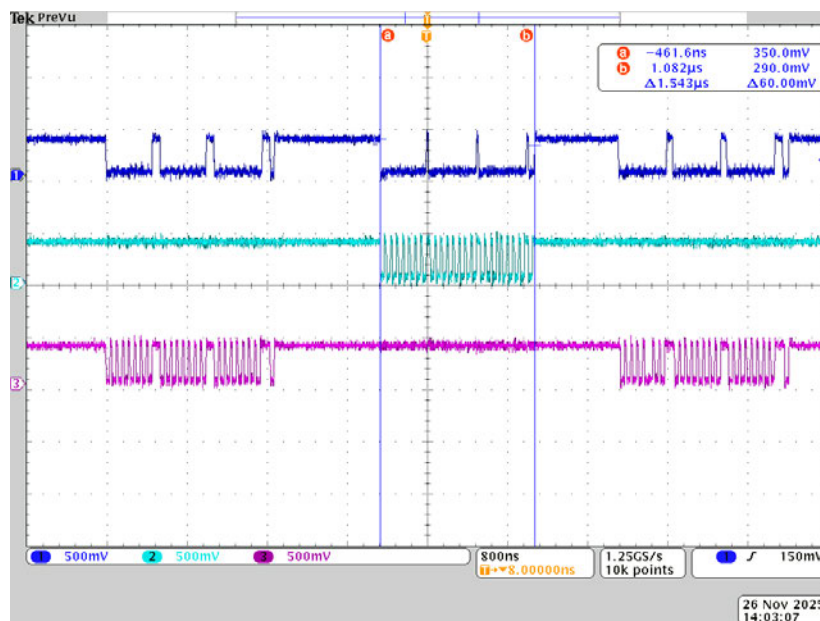


Abbildung 5.2: Signalverlauf während eines Schreib- und Lesevorgangs über EMIFA (EDMA)

Im Vergleich zur sequenziellen Zugriffsvariante über den DSP-Core bietet die DMA-basierte Übertragung eine deutlich geringere Latenz bei größeren Datenblöcken. Wie in

Abschnitt 5.3.1 erläutert, führt der DSP-Core-basierte Zugriff zu systematischen Leerlaufphasen zwischen einzelnen Transaktionen. Die Verwendung des EDMA-Controllers reduziert diese Zwischenpausen erheblich, da die Datenübertragung als zusammenhängende Burst-Sequenzen ohne DSP-Core-Interaktion erfolgt [33]. Ein Großteil der durch den DSP-Core gesteuerten Einzeltransaktionen verursachten Turnaround- und Leerlaufphasen entfällt dadurch. Die maximale effektive Blocklänge wird hierbei jedoch durch die interne Pufferarchitektur der EDMA-/EMIFA-Schnittstelle sowie durch die Transaktionssegmentierung begrenzt, wodurch längere Transfers weiterhin in mehrere aufeinanderfolgende Zugriffsblöcke unterteilt werden müssen. Pro Übertragungsblock können in der hier verwendeten Konfiguration jeweils acht Datenwörter zusammenhängend übertragen werden. Nach Abschluss eines solchen Blocks muss der Bus (beim Lesezugriff ca. 80 ns und beim Schreibzugriff ca. 20 ns) pausieren, bevor der nächste Datenblock geladen und übertragen werden kann [33]. Dieses Verhalten ist in Abbildung 5.2 gut zu erkennen.

Trotz dieser architekturbedingten Grenzen zeigt sich eine signifikant höhere Busauslastung und eine deutlich verbesserte Gesamtdatenrate im Vergleich zur DSP-Core-gesteuerten Zugriffsmethode [34]. Die EDMA-basierte Kommunikation erwies sich daher als bevorzugte Methode für latenzkritische und datenintensive Verarbeitungsschritte im Rahmen des DSP-FPGA-Systems.

Zugriffsart	DSP-Core	EDMA	Differenz t_{diff} [µs]
Schreibzugriff t_S [µs]	8	6.24	1.76
Lesezugriff t_L [µs]	24	6.96	17.04

Tabelle 5.2: Vergleich der Zugriffszeiten bei 100 Datenwörtern über EMIFA mittels DSP-Core vs. EDMA (gemessen per Oszilloskop)

Ein quantitativer Vergleich der für 100 Datenwörter benötigten Übertragungszeiten ist in Tabelle 5.2 dargestellt. Die angegebenen Werte basieren teilweise auf direkten Oszilloskopmessungen sowie auf den folgenden Berechnungen:

$$\text{DSP-Core:} \quad t_S = N_{\text{Datenwörter}} \cdot t_{\text{EMIFA}} + N_{\text{Datenwörter}} \cdot t_{S1_p} \quad (5.1)$$

$$\text{EDMA:} \quad t_S = N_{\text{Datenwörter}} \cdot t_{\text{EMIFA}} + (N_{\text{Datenwörter}} \bmod 8) \cdot t_{S2_p} \quad (5.2)$$

$$\text{DSP-Core:} \quad t_L = N_{\text{Datenwörter}} \cdot t_{\text{EMIFA}} + N_{\text{Datenwörter}} \cdot t_{L1_p} \quad (5.3)$$

$$\text{EDMA:} \quad t_L = N_{\text{Datenwörter}} \cdot t_{\text{EMIFA}} + (N_{\text{Datenwörter}} \bmod 8) \cdot t_{L2_p} \quad (5.4)$$

Dabei bezeichnet $N_{\text{Datenwörter}}$ die Anzahl der zu übertragenden Datenwörter. Die in den Formeln enthaltenen Pausenzeiten $t_{S1_p}, t_{S2_p}, t_{L1_p}, t_{L2_p}$ wurden empirisch aus den Oszilloskopmessungen abgeleitet. Die Formeln verdeutlichen den Effizienzgewinn, der durch den Einsatz des EDMA-Controllers gegenüber sequenziellen DSP-Zugriffen erzielt wird.

5.3.3 Pufferung und Datenpfad

Die in Abschnitt 4.7 vorgestellte Datenflussarchitektur wird auf DSP-Seite durch eine speicherbasierte Zwischenpufferung der OFDM-Frames realisiert. Der Übertragungskanal wird dabei modellhaft als abstrakter Zwischenspeicher interpretiert, in dem jeweils ein vollständiger Frame temporär abgelegt wird. Diese Modellierung erlaubt eine klare Trennung zwischen Sender- und Empfängerpfad, da die inverse Verarbeitung auf Empfängerseite unmittelbar aus diesem Puffer erfolgt. Wie bereits im Grundlagen- und Konzeptkapitel erläutert, stellt die physikalische Modellierung des Übertragungskanals nicht den Schwerpunkt dieser Arbeit dar. Aus diesem Grund wurden Module zur Kanalschätzung, Kanalverzerrung sowie zur Taktrückgewinnung nicht implementiert. Der Kanal wird somit idealisiert betrachtet, und es erfolgt keine Berücksichtigung realer Stör- oder Verzerrungseffekte.

Die Kommunikation zwischen ADC/DAC, DSP und FPGA basiert auf einer Kombination aus Interrupt-gesteuerten Ereignissen sowie kontrollierten EMIFA-Transfers.

ADC- und DAC-Interrupt-Service-Routine

Die Audio-I/O erfolgt mit einer festen Abtastrate von 100 kHz. Für den ADC wird ein Ringpuffer (`IN_FIFO`) verwendet, in den bei jedem Interrupt eine neue Abtastwert geschrieben wird. Sobald 12 Werte gesammelt wurden, wird eine globale Statusvariable gesetzt, die den Start der weiteren Verarbeitung bzw. der Übertragung des aktuellen Blocks auslöst [1].

Der DAC arbeitet nach dem gleichen Prinzip: Eine Ausgabe erfolgt nur dann, wenn ein vollständiger OFDM-Frame empfangsseitig vollständig dekodiert und in den Ausgabepuffer (`OUT_FIFO`) überführt wurde. Im Idealfall entsprechen die am DAC ausgegebenen Werte exakt bis auf Quantisierungs- und Rundungsfehler den am ADC erfassten Eingangssamples, lediglich um die systembedingte Verarbeitungslatenz zeitlich versetzt.

Abbildung 5.3 zeigt das zugrunde liegende Ablaufdiagramm der Interrupt-Service-Routinen für ADC und DAC.

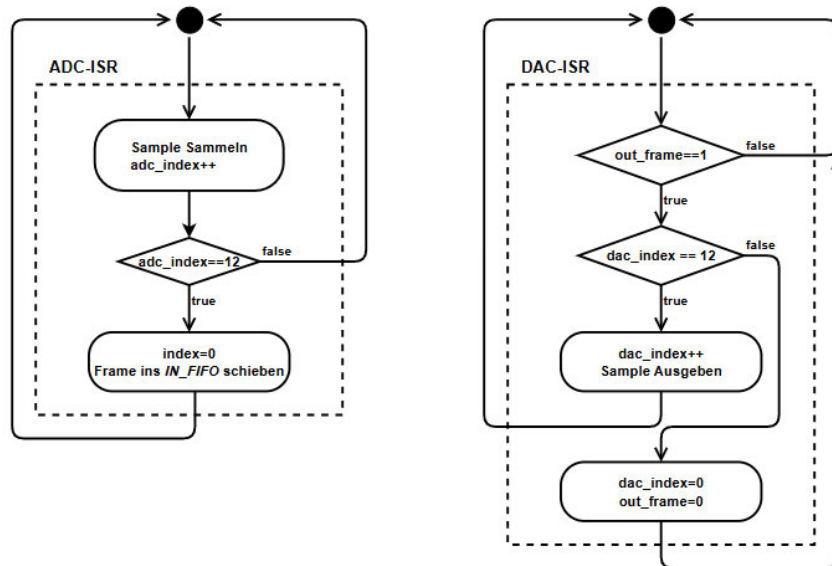


Abbildung 5.3: Ablaufdiagramm der ADC- und DAC-Interrupt-Service-Routinen

Ablauflogik der OFDM-Verarbeitung auf der DSP-Seite

Die eigentliche Datenübertragung zwischen DSP und FPGA erfolgt – wie in Abschnitt 5.3.2 beschrieben – blockweise mittels EDMA über den EMIFA-Bus. Die Entscheidung zugunsten dieser Übertragungsart wurde bewusst getroffen, da sie eine entkoppelte, effizientere Kommunikation zwischen den beiden Komponenten ermöglicht und gleichzeitig den DSP-Core von lauffzeitkritischen Steuer- oder Kopieroperationen entlastet. Auf dem DSP wurden keine signalverarbeitenden Komponenten des OFDM-Systems implementiert. Der DSP übernimmt stattdessen ausschließlich die übergeordnete Steuerung sowie die koordinierte Initialisierung der EDMA-Transfers, über welche die Datenframes blockweise an den FPGA weitergeleitet werden. Die vollständige Signalverarbeitung erfolgt auf der FPGA-Seite, wodurch der DSP als schlanker Steuerknoten agiert.

Die Verarbeitung eines kompletten OFDM-Frames folgt der in Abbildung 5.4 dargestellten Ablauflogik. Nach dem vollständigen Füllen des Eingangspuffers durch die ADC-Interrupt-Service-Routine wird per globalem Steuerflag die Übertragung an den TX-Pfad des FPGA ausgelöst. In einem ersten Schritt prüft der DSP, ob der TX-Pfad verfügbar ist, bevor die sequentielle Übertragung der 12 Eingangswerte über den EMIFA-Bus erfolgt.

Nach Abschluss der Übertragung wird durch gezielten Schreibzugriff auf ein Steuerregister das **GO**-Bit gesetzt, wodurch die Verarbeitung auf FPGA-Seite aktiviert wird. Anschließend überwacht der DSP im Polling-Verfahren das zugehörige **DONE**-Bit. Sobald dieses gesetzt ist, signalisiert der FPGA die erfolgreiche Verarbeitung und Bereitstellung des Ausgangsframes. Der DSP löscht das **DONE**-Bit durch einen weiteren Schreibzugriff und liest den berechneten Frame über den EDMA aus dem Ausgabepuffer des FPGA in ein temporäres Speicherarray ein.

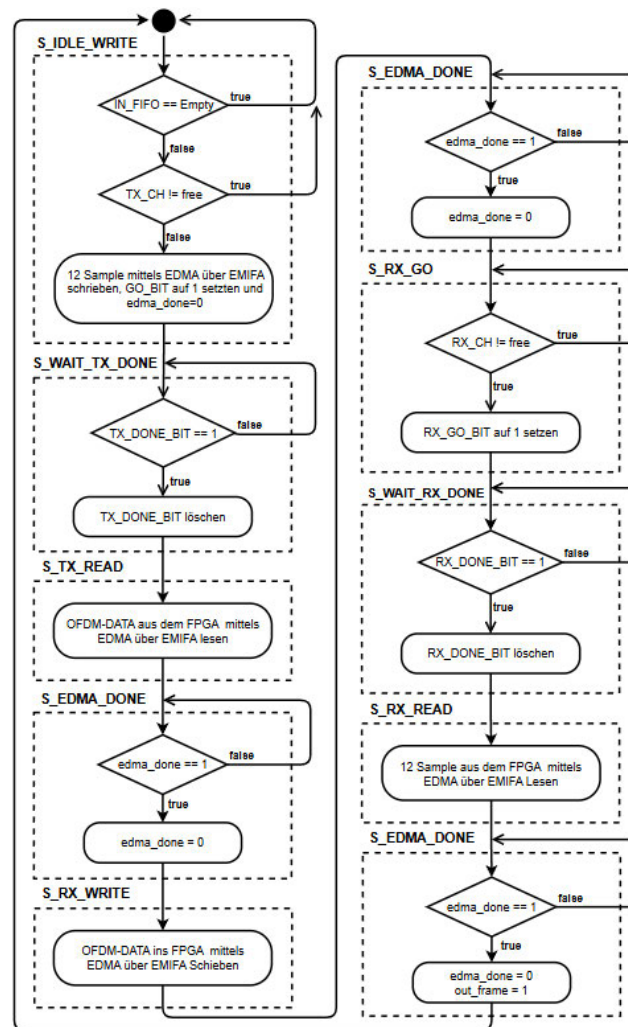


Abbildung 5.4: Ablaufdiagramm der OFDM-Verarbeitung zwischen DSP und FPGA

Die empfangsseitige Weiterverarbeitung erfolgt analog: Nach Prüfung der Verfügbarkeit des RX-Pfades wird der empfangene Frame erneut über den EDMA an den FPGA übertragen. Auch hier wird die Verarbeitung durch Setzen des entsprechenden **GO**-Bits gestartet. Nach Abschluss signalisiert das **DONE**-Bit die Verfügbarkeit der berechneten Ausgabedaten, die anschließend in den Ausgabepuffer **OUT_FIFO** übernommen werden. Im Anschluss wird eine globale Statusvariable gesetzt. Diese dient als Quelle für die DAC-Interrupt-Service-Routine, die die kontinuierliche Ausgabe an den Digital-Analog-Wandler übernimmt.

5.4 FPGA-seitige Systemintegration

Dieser Abschnitt beschreibt die Implementierung der EMIFA-Schnittstelle sowie die Integration der verarbeitungsspezifischen HLS-IP-Cores auf der FPGA-Seite. Die Architekturkonzeption basiert auf den im Kapitel 4 formulierten Systementscheidungen und wurde im Projektverlauf iterativ weiterentwickelt. Dabei erfolgte eine fortlaufende funktionale Validierung der Einzelschritte, um eine schrittweise Absicherung der Gesamtsystemfunktionalität zu gewährleisten [52].

Ein zentraler Aspekt der Implementierung ist die saubere Trennung zwischen der EMIFA-Schnittstelle und der FPGA-internen Verarbeitungslogik. Beide Komponenten operieren in unterschiedlichen Taktdomänen: Während das EMIFA-Interface synchron zum externen EMIFA-Takt arbeitet, sind die HLS-Verarbeitungskerne vollständig auf den internen FPGA-Systemtakt abgestimmt. Zur sicheren Kommunikation über Domänengrenzen hinweg wurden Synchronisierungseinheiten sowie Puffermechanismen implementiert, um Metastabilität und Timingverletzungen systematisch zu vermeiden.

Die nachfolgenden Abschnitte gehen detailliert auf die Implementierung des EMIFA-Interfaces (Abschnitt 5.4.1), die Anbindung an AXI4-Streams (Abschnitt 5.4.2), sowie die Integration und Parametrierung der HLS-basierten Signalverarbeitungsblöcke (Abschnitt 5.5) ein.

5.4.1 EMIF-Interface und FIFO-Anbindung

Für die Verbindung zwischen DSP und FPGA wurde ein speziell auf den EMIFA-Bus des TMS320C6747 abgestimmtes Interface entworfen [4]. Ziel war die zuverlässige Übertragung von Daten gemäß den funktionalen Anforderungen aus Abschnitt 3.3. Die Umsetzung folgt dem in Abschnitt 4.4.3 beschriebenen synchronen Kommunikationsmodell.

Die Architektur nutzt sowohl das vom DSP bereitgestellte Taktsignal **EMIFA_CLK** als auch den internen FPGA-Systemtakt **FPGA_CLK**. Die Steuerlogik für das EMIFA-Protokoll arbeitet vollständig im EMIFA-Taktdomänenbereich. Zur sicheren Kommunikation mit den asynchron getakteten Verarbeitungsmodulen auf der FPGA-Seite wurden bestimmte Takt-Domänen-Synchronisierer implementiert. Abbildung 5.5 zeigt die entsprechende Domänentrennung und das Schnittstellenkonzept.

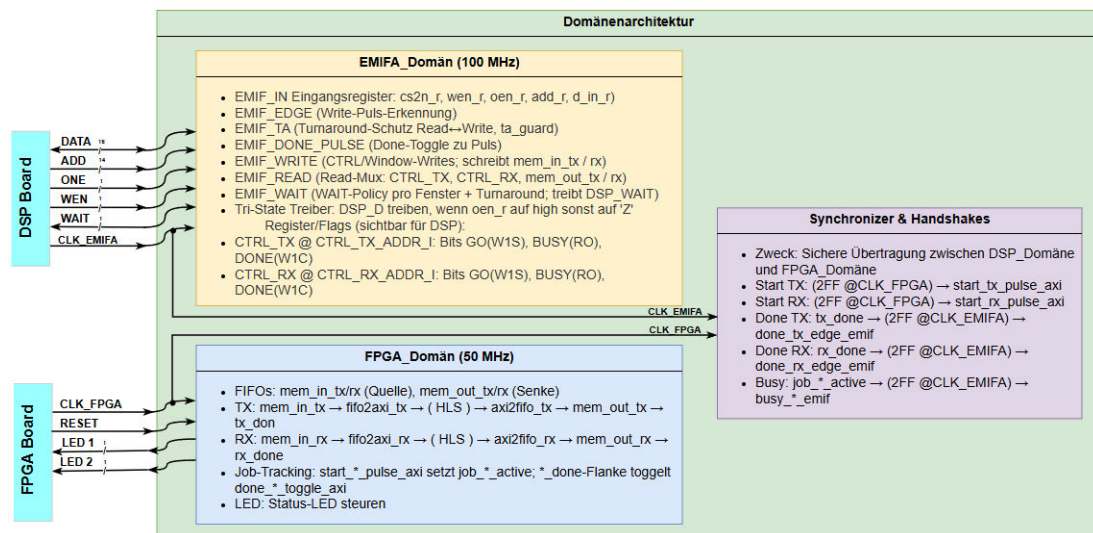


Abbildung 5.5: Takt- und Domänenarchitektur des EMIF-Interfaces

Die Auswertung aller EMIFA-Daten- und Steuersignale erfolgt synchron zum EMIFA-Takt. Eingehende Daten werden in den Eingabepuffer **FIFO_IN**, ausgehende Daten aus dem Ausgabepuffer **FIFO_OUT** geschrieben bzw. gelesen. Statussignale wie **Start** und **Busy** werden domänenübergreifend mittels Synchronisierern übertragen, um Race Conditions und Metastabilitäten zuverlässig zu vermeiden.

Die verarbeitungsspezifischen Einheiten – realisiert als HLS-basierte IP-Cores – arbeiten vollständig im FPGA-Systemtakt (**FPGA_CLK**). Der Datenfluss ist strikt strukturiert: Der HLS-Core liest die Eingangsdaten sequenziell aus **FIFO_IN**, verarbeitet diese und schreibt die Ergebnisse in **FIFO_OUT** zurück. Alle Schnittstellen sind gemäß den AXI4-Stream-Spezifikationen implementiert [47]. Das EMIF-Interface unterstützt dabei ausschließlich unidirektionale Transaktionen: In jeder Verarbeitungseinheit (TX- oder RX-Pfad) kann zu einem Zeitpunkt nur gelesen oder geschrieben werden – nicht beides gleichzeitig. Parallel dazu können, abhängig vom Status der Verarbeitungslogik, vorbereitende Steuerzugriffe auf dem jeweils freien Pfad erfolgen, sofern dieser aktuell keine aktiven Datenübertragungen durchführt, oder bereits Daten bereitgestellt hat [3].

Die FIFO-Architektur wurde so ausgelegt, dass sie den spezifischen Anforderungen der TX- und RX-Pfade entspricht. Die zugrunde liegende Pufferstrategie folgt strikt der Datenflussrichtung. Abbildung 4.9 visualisiert das Gesamtkonzept.

Für den Sendepfad (TX) kommen zwei FIFO-Strukturen zum Einsatz:

- **Eingangs-FIFO** ($\text{FIFO}_{\text{in,TX}}$): Puffert die vom DSP übertragenen Nutzdaten. Ihre Tiefe ergibt sich aus:

$$\text{FIFO}_{\text{in,TX}} = \frac{N_{\text{Daten}} \cdot \log_2(M_{\text{QAM}})}{16 \text{ Bit}}, \quad (5.5)$$

wobei N_{Daten} die Anzahl der Modulationssymbole und M_{QAM} die verwendete Modulationsordnung beschreibt. Die Division durch 16 Bit ergibt sich aus der internen Wortbreite des EMIFA-Busses.

- **Ausgangs-FIFO** ($\text{FIFO}_{\text{out,TX}}$): Speichert die nach IFFT und CP-Ergänzung erzeugte Zeitfolge. Die notwendige FIFO-Tiefe beträgt:

$$\text{FIFO}_{\text{out,TX}} = N + N_{\text{CP}}, \quad (5.6)$$

wobei N die IFFT-Länge und N_{CP} die Anzahl der Cyclic Prefix darstellt.

Für den Empfangspfad (RX) gelten die gleichen FIFO-Größen – jedoch in umgekehrter Reihenfolge:

$$\text{FIFO}_{\text{in,RX}} = \text{FIFO}_{\text{out,TX}} \quad \text{und} \quad \text{FIFO}_{\text{out,RX}} = \text{FIFO}_{\text{in,TX}}. \quad (5.7)$$

Diese symmetrische Zuordnung gewährleistet, dass für beide Richtungen exakt auf die OFDM-Verarbeitung abgestimmte Puffergrößen zur Verfügung stehen.

Zur Analyse der Verarbeitungszeiten wurde das Interface um einen laufenden Taktzähler erweitert, der für TX- und RX-Pfad separat gestartet, gestoppt und ausgelesen werden kann. Die Ansteuerung erfolgt über definierte Registerbereiche. Damit lassen sich exakte Messwerte zur Dauer eines vollständigen Verarbeitungslaufs erfassen, was gezielte Performanzanalysen erlaubt.

Zusätzlich wurden zwei Status-LEDs zur Laufzeitvisualisierung integriert: Eine LED zeigt den aktiven Zustand des EMIF-Interfaces an, während die zweite den Verarbeitungstakt des HLS-Cores abbildet. Letztere wird bei aktiver Verarbeitung sequenziell getoggelt – die resultierende Blinkfrequenz entspricht somit der inneren Taktung der HLS-Verarbeitung und erlaubt eine grobe Einschätzung der Pipelineauslastung.

5.4.2 AXI4-Stream: FIFO $\leftrightarrow$ AXI

Zwischen dem EMIF-Interface und dem HLS-basierten IP-Core befinden sich zwei zentrale Vermittlungsmodule, die den Übergang zwischen den synchronen FIFO-Speichern und der AXI4-Stream-Schnittstelle realisieren. Diese Komponenten übernehmen die Protokollanpassung und die temporale Steuerung des Datenflusses in beiden Richtungen:

- **FIFO $\rightarrow$ AXI-Stream:** Überträgt eingehende Daten aus dem Eingangspuffer `FIFO_IN` an den HLS-Core. Die Übertragung startet, sobald ein gültiges Startsignal (**GO**) vorliegt.
- **AXI-Stream $\rightarrow$ FIFO:** Empfängt die vom HLS-Core generierten Ausgabedaten und speichert sie geordnet in `FIFO_OUT`. Nach Abschluss des Transfers wird das **DONE**-Signal zur Signalisierung des Verarbeitungsendes gesetzt.

Beide Module wurden als eigenständige Steuerwerke realisiert und jeweils durch ein ASMD-Diagramm (Algorithmic State Machine with Data Path) spezifiziert. Die korrekte Funktion wurde über Simulationen in AMD Vivado verifiziert und in das Gesamtprojekt integriert.

Die Synchronisation mit den FIFO-Schnittstellen erfolgt über Handshake-Signale wie **valid**, **ready** und **last**, entsprechend den AXI4-Stream-Spezifikationen. Die Verarbeitung erfolgt strikt sequenziell, wobei interne Zähler den Fortschritt des Transfers kontrollieren und die korrekte Anzahl an Datenwörtern garantieren.

Insgesamt stellen die beiden Module eine flexible und erweiterbare Brücke zwischen der externen EMIF-Anbindung und der internen HLS-Verarbeitung dar. Ihre Modularität erlaubt eine Wiederverwendung auch in künftigen Designs mit ähnlicher Schnittstellenstruktur.

FIFO zu AXI4-Stream

Das Modul *FIFO zu AXI4-Stream* übernimmt die Aufgabe, Daten sequenziell aus dem Eingangspuffer auszulesen und gemäß dem AXI4-Stream-Protokoll an den HLS-Core weiterzuleiten. Die Implementierung basiert auf dem in Abschnitt 2.6 dargestellten AXI-Stream-Schema. Der zugrunde liegende Steuerablauf ist als ASMD-Diagramm in Abbildung 5.6 dargestellt [13].

Die Zustandsmaschine besteht aus vier Zuständen:

- **S0_Idle**: Wartet auf ein aktives **START**-Signal.
- **S1_Wait**: Wartet einen Takt, bis gültige FIFO-Ausgabedaten bereitstehen (bedingt durch BRAM-Latenz).
- **S2_SendWord**: Legt das aktuelle Datenwort auf den AXI-Datenbus, setzt **TVALID** und prüft, ob es sich um das letzte Wort handelt (**TLAST**).
- **S3_WaitForReady**: Wartet auf ein aktives **TREADY**-Signal vom Empfänger, bevor zum nächsten Wort übergegangen wird.

Wird das **START**-Signal gesetzt, beginnt der Datenfluss, und das **BUSY**-Signal wird aktiviert. Aufgrund der FIFO-inhärenten Latenz erfolgt zunächst ein Taktdelay im Zustand **S1_Wait**. Anschließend beginnt im Zustand **S2_SendWord** die Übertragung, bei der pro Datenwort **TVALID** gesetzt wird. Im letzten Takt wird zusätzlich **TLAST** aktiviert, um das Ende der Übertragung zu kennzeichnen. Der Wechsel zurück in den Leerlaufzustand erfolgt, sobald **TREADY** den erfolgreichen Empfang signalisiert.

Nach Abschluss des Transfers wird das **BUSY**-Signal deaktiviert, und für einen Takt wird das **DONE**-Signal gesetzt, das den Abschluss der Transaktion anzeigt. Dieser Ablauf wurde durch Simulation validiert.

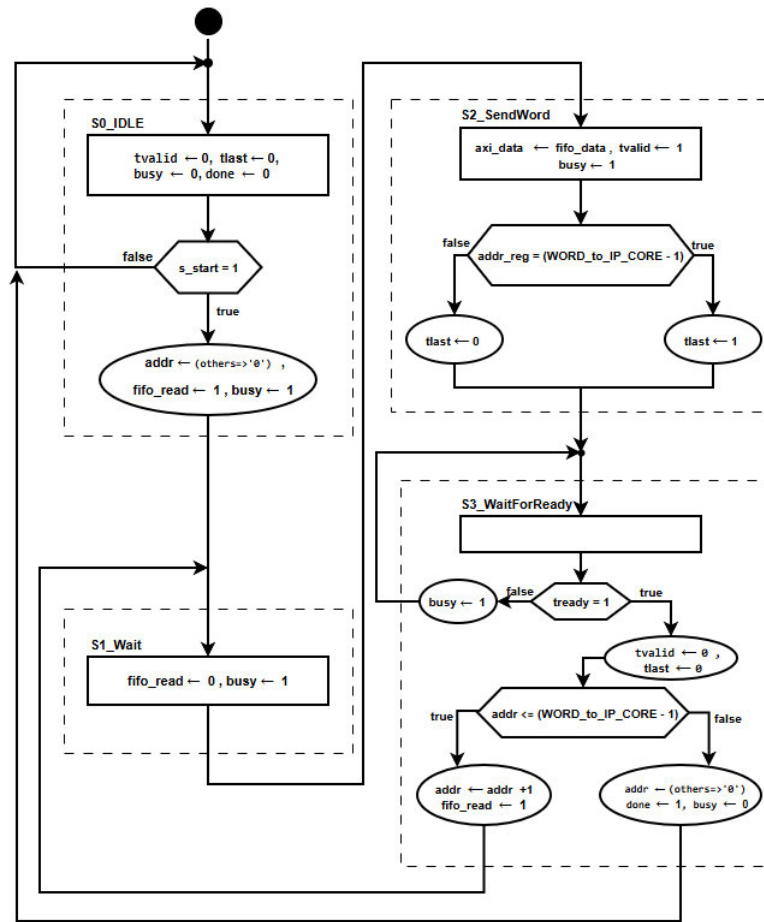


Abbildung 5.6: ASMD-Diagramm des FIFO-zu-AXI4-Stream-Moduls

Die Simulationswform in Abbildung 5.7 zeigt den zeitlichen Ablauf eines vollständigen Transfers. Nach dem Setzen des **START**-Signals wird das **BUSY**-Signal aktiviert, und die Datenübertragung beginnt. Nach dem letzten Wort folgt das kurzzeitige Setzen von **DONE**.

Die Simulation beinhaltet zudem ein Szenario, in dem das **TREADY**-Signal vom Empfänger temporär deaktiviert wird (in der Simulationswform zwischen ca. 200 ns und 300 ns). Das Modul reagiert korrekt, indem es den aktuellen Datensatz zurückhält und erst bei Reaktivierung von **TREADY** mit der Übertragung fortfährt. Jeder Datenwort-transfer erfordert in der dargestellten Sequenz drei Takte.

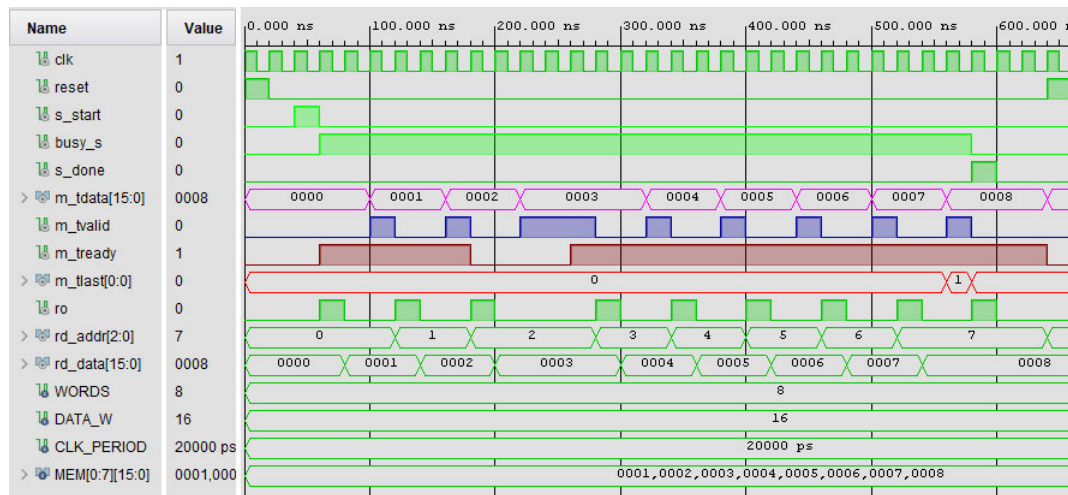


Abbildung 5.7: Simulationswellenform der FIFO-zu-AXI4-Stream Übertragung

AXI4-Stream zu FIFO

Das zweite Modul bildet die Gegenrichtung ab: Es empfängt die vom HLS-Core bereitgestellten Daten über die AXI4-Stream-Schnittstelle und schreibt diese sequenziell in den Ausgangspuffer `FIFO_OUT`. Die Steuerung der Datenaufnahme erfolgt protokollkonform auf Basis der Handshake-Signale **TVALID**, **TREADY** und **TLAST** [13].

Der Steuerablauf wurde über ein ASMD-Diagramm mit zwei Zuständen modelliert (siehe Abbildung 5.8):

- **S0_Idle**: Wartet auf ein aktives **TVALID**-Signal des HLS-Cores. Liegen gültige Daten an, wird **TREADY** aktiviert, und die FSM wechselt in den Datenempfangszustand.
- **S1_Stream**: Bei gleichzeitig aktiven Signalen **TVALID** und **TREADY** wird das aktuelle Datenwort übernommen und in den FIFO geschrieben. Nach jedem Transfer erfolgt eine Prüfung auf das **TLAST**-Signal bzw. die Erreichung der erwarteten Datenwortanzahl. In beiden Fällen wird der Empfangsvorgang abgeschlossen.

Nach Beendigung der Übertragung wird das **TREADY**-Signal deaktiviert, das **DONE**-Signal für einen Takt gesetzt, und die FSM kehrt in den Leerlaufzustand zurück. Abbildung 5.8 zeigt das zugehörige ASMD-Diagramm, während Abbildung 5.9 die funktionale Verifikation per Simulation darstellt.

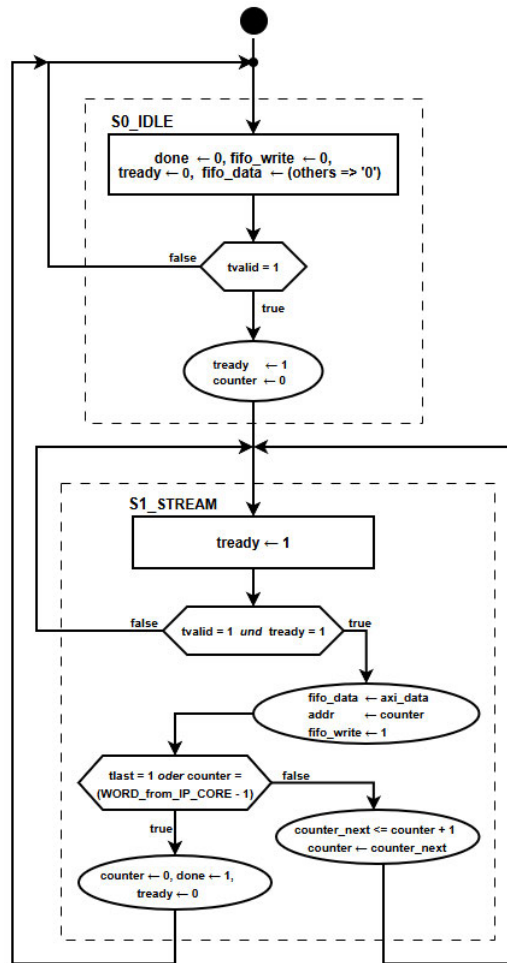


Abbildung 5.8: ASMD-Diagramm des AXI4-Stream-zu-FIFO-Moduls

Die Simulationswveform bestätigt das erwartete Verhalten: Bei aktivem **TVALID** und gesetztem **TREADY** werden die Daten sequentiell übernommen und korrekt in den FIFO geschrieben. Nach Erkennung des letzten Datenworts (**TLAST** = 1) wird der Übertragungsvorgang ordnungsgemäß abgeschlossen, und das **DONE**-Signal für einen Takt aktiviert. Das Verhalten entspricht in Struktur und Timing dem des FIFO-zu-AXI-Moduls, erfolgt jedoch in umgekehrter Datenrichtung.

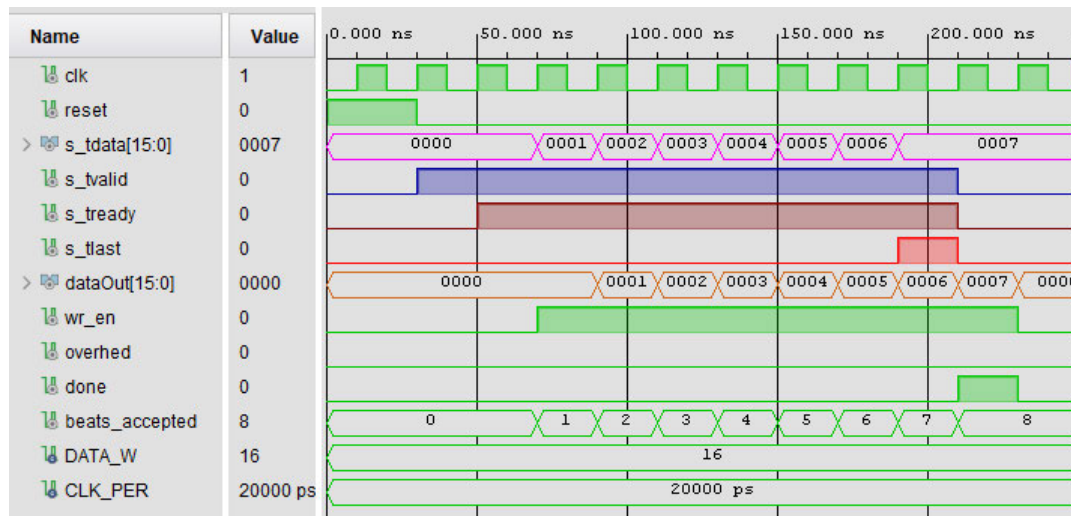


Abbildung 5.9: Simulationswellenform des AXI4-Stream-zu-FIFO-Moduls

5.4.3 Integration der Funktionsmodule

In diesem Abschnitt wird die Integration der zuvor entwickelten Untermodule beschrieben. Dazu zählen insbesondere das EMIFA-Interface, die Takt- und Domärentrennung, die AXI4-Vermittlungsmodule (**FIFO-zu-AXI4-Stream** und **AXI4-Stream-zu-FIFO**) sowie die zugehörigen Synchronisationsmechanismen. Ziel ist die Validierung einer durchgängigen Datenübertragung über alle beteiligten Verarbeitungspfadkomponenten hinweg.

Für erste Integrationstests wurden die AXI-Module intern rückgekoppelt (Loopback-Konfiguration), sodass eingehende Daten ohne Verarbeitung direkt vom Eingangspuffer über das AXI4-Stream-Protokoll in den Ausgangspuffer weitergeleitet wurden. Abbildung 5.10 zeigt das vereinfachte Testsetup, bei dem die Verbindung zwischen **FIFO_zu_AXI** und **AXI_zu_FIFO** innerhalb der FPGA-Logik – ohne Zwischenschaltung des HLS-IP-Cores – erfolgt.

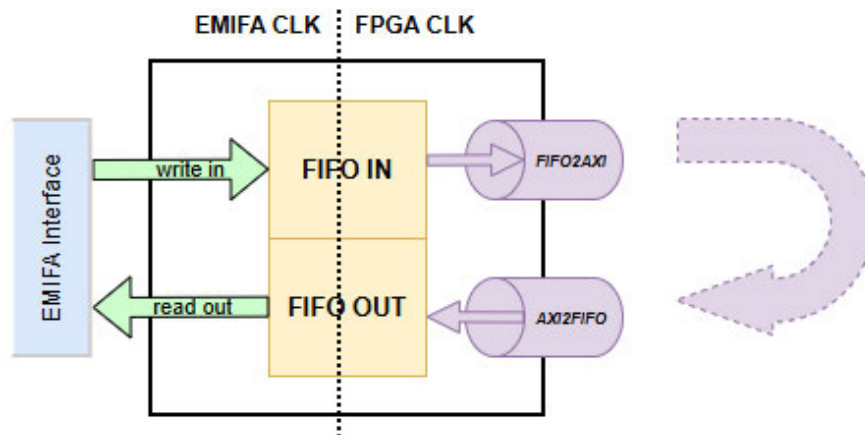


Abbildung 5.10: Interner Loopback zur Integration der FIFO-AXI-Module

Der Testablauf folgte einem fest definierten Muster:

1. Auf DSP-Seite wurden N Werte in den FIFO-Eingangspuffer geschrieben.
2. Die Übertragung wurde durch Setzen des **GO**-Bits gestartet.
3. Das Modul **FIFO zu AXI** übermittelte die Daten sequenziell über die AXI4-Stream-Schnittstelle.
4. Das Modul **AXI zu FIFO** empfing die Daten und speicherte sie im Ausgangspuffer **FIFO_OUT**.
5. Nach erfolgreicher Übertragung wurde das **DONE**-Signal aktiviert.
6. Der DSP las die Werte aus dem Ausgangspuffer und validierte sie gegen die ursprünglichen Eingabedaten.

Die Tests bestätigten sowohl die korrekte Funktion der Taktdomänentrennung als auch die Integrität der übertragenen Daten. Insbesondere konnte gezeigt werden, dass:

- alle Status- und Steuersignale sicher domänenübergreifend übertragen wurden,
- keine Datenverluste oder Duplikate auftraten,
- das **DONE**-Signal den Abschluss der Übertragung exakt anzeigte.

Die erfolgreiche Durchführung der Integrationstests bildete die Grundlage für die spätere Einbindung des HLS-basierten Verarbeitungsmoduls.

Ergänzend zur funktionalen Integration wurde auch die Adressierung der FPGA Komponenten auf DSP-Seite eindeutig definiert. Die Adressräume der vier FIFO-Speicherbereiche (TX und RX, jeweils IN/OUT) sowie der zugehörigen Steuer- und Zeitregister wurden über `#define`-Direktiven im DSP-Code statisch festgelegt. Dadurch ist ein konsistenter, hardwarekompatibler Zugriff sichergestellt. Das zugehörige Mapping ist in Listing 5.3 dargestellt.

```

1  /*****
2  EMIFA/FPGA address map & control bits (word addressed)
3  *****/
4  #define TX_IN_BASE    0x0000 /* reserviert bis 0x07FF für FIFO_IN_TX */
5  #define TX_OUT_BASE   0x0800 /* reserviert bis 0x0FFF für FIFO_OUT_TX */
6  #define RX_IN_BASE    0x1000 /* reserviert bis 0x17FF für FIFO_IN_RX */
7  #define RX_OUT_BASE   0x1800 /* reserviert bis 0x1FFF für FIFO_OUT_RX */
8
9  /* Steuer- und Timing-Register (oberster Bereich des 14-Bit Adressraums) */
10 #define CTRL_TX        0x3FFF /* Steuerregister für TX-Pfad */
11 #define CTRL_RX        0x3FFE /* Steuerregister für RX-Pfad */
12 #define TIME_TX        0x3FFD /* Zeitmess-Register für TX-Pfad */
13 #define TIME_RX        0x3FFC /* Zeitmess-Register für RX-Pfad */
14
15 /* Bitmasken für das Steuerregister */
16 #define CTRL_GO        0x0001 /* Start-Bit: Löst die Verarbeitung im jeweiligen Pfad
    aus */
17 #define CTRL_BUSY      0x0002 /* Busy-Bit: Zeigt an, ob der Pfad aktuell beschäftigt
    ist */
18 #define CTRL_DONE      0x0004 /* Done-Bit: Signalisiert, dass verarbeitete Daten
    bereitstehen */

```

Listing 5.3: Adresszuweisung für FIFO- und Steuerregister im EMIFA-Adressraum

Die Steuer- und Zeitregister befinden sich im obersten Bereich des verfügbaren 14-Bit-Adressraums, und sind den jeweiligen Pfad eindeutig zugeordnet. Dadurch kann der DSP gezielt Zustände abfragen (z. B. **BUSY**, **DONE**) oder Operationen starten (durch Setzen des **GO**-Bits).

Die Größe der reservierten FIFO-Adressbereiche wurde so gewählt, dass sie auch für maximale Datenmengen ausreichend dimensioniert sind. Bei einer beispielhaften FFT-Länge von $N = 256$ und einem Cyclic Prefix von $N_{CP} = 32$ ergibt sich gemäß Formel 5.6

eine notwendige FIFO-Tiefe:

$$\text{FIFO}_{\text{in,TX}} = N + N_{\text{CP}} = 288 \quad (5.8)$$

Mit einer Reserve von 2048 Worten pro FIFO ist damit ein großzügiger Adressraum vorgesehen. Auf Hardwareseite wird die tatsächliche FIFO-Tiefe bedarfsgerecht implementiert, während der DSP nur den relevanten Bereich nutzt. Diese Trennung erlaubt eine flexible Ressourcennutzung ohne Adresskonflikte oder Speicherverschwendung.

Die konsistente Adresszuweisung bildet die Grundlage für eine konfliktfreie Kommunikation zwischen Software und Hardware und wird sowohl in der manuellen Steuerung als auch bei EDMA-gestützten Transfers verwendet.

5.5 High-Level-Synthese der Verarbeitungskette

In diesem Abschnitt wird die FPGA-seitige Umsetzung der OFDM-Verarbeitungskette unter Verwendung von High-Level-Synthese (HLS) beschrieben. Die funktionalen Grundlagen sowie die zugrunde liegenden Entwurfsentscheidungen wurden bereits in Kapitel 4 erläutert. Die praktische Implementierung basiert auf den in Abschnitt 2.3 beschriebenen HLS-Mechanismen und nutzt die dort vorgestellten Optimierungsstrategien sowie Schnittstellenkonzepte [15].

Ziel war es, die vollständige Signalverarbeitung innerhalb der FPGA-Architektur mithilfe von HLS abzubilden und dabei eine optimale Balance zwischen Ressourcenverbrauch, minimaler Latenz, maximaler Taktfrequenz und Integrationsfähigkeit in das Gesamtsystem zu erzielen. Dabei wurden sowohl algorithmische als auch architekturbezogene Aspekte berücksichtigt, insbesondere Pipelining, Datenflusssteuerung und Portdefinitionen für Streaming-Schnittstellen [46].

5.5.1 Schnittstellen und Steuerung

Die Kommunikation zwischen den HLS-basierten Verarbeitungsmodulen und der umgebenden FPGA-Peripherie erfolgt über die AXI4-Stream-Schnittstellen, wie in Abschnitt 2.6 beschrieben. Diese Schnittstellen ermöglichen einen kontinuierlichen, taktsynchronen Datenstrom zwischen den IP-Cores und den Vermittlungsmodulen `FIFO` ↔ `AXI4-Stream` (vgl. Abschnitt 5.4.2).

Die Eingangsdaten der beiden Funktionsmodule – `ofdm_tx` (Sender) und `ofdm_rx` (Empfänger) – werden im 16-Bit-Format als `unsigned integer` übermittelt. Die Ausgabedaten liegen im Q1.15-Fixpunktformat mit Vorzeichenbit und 15 Nachkommastellen vor und werden durch den Typ `ap_fixed<16,1>` repräsentiert [39].

Ein exemplarischer Ausschnitt der relevanten Portdefinition ist in Listing 5.4 dargestellt:

```

1  #include <hls_stream.h>
2  #include <ap_fixed.h>
3
4  /* Eingangsstream (16 Bit unsigned) */
5  typedef ap_axiu<16,0,0,0> axis16_t;
6  typedef hls::stream<axis16_t> in_axis16_stream;
7
8  /* Ausgangsstream (Q15: 1 Vorzeichenbit, 15 Bit Mantisse) */
9  typedef ap_fixed<16, 1, AP_RND, AP_SAT> q15_1_t;
10 typedef hls::axis<q15_1_t, 0, 0, 0> q15_axis_t;
11 typedef hls::stream<q15_axis_t> out_axisQ15_stream;
12
13 /* OFDM Sender Module */
14 void ofdm_tx(in_axis16_stream &DATA_IN_TX, out_axisQ15_stream &DATA_OUT_TX, ap_uint
    <1> &led) {
15     #pragma HLS INTERFACE axis          port=DATA_IN_TX
16     #pragma HLS INTERFACE axis          port=DATA_OUT_TX
17     #pragma HLS INTERFACE ap_none       port=led
18     #pragma HLS INTERFACE ap_ctrl_none  port=return
19     // ...
20 }
21
22 /* OFDM Empfänger Module */
23 void ofdm_rx(in_axis16_stream &DATA_IN_RX, out_axisQ15_stream &DATA_OUT_RX) {
24     #pragma HLS INTERFACE axis          port=DATA_IN_RX
25     #pragma HLS INTERFACE axis          port=DATA_OUT_RX
26     #pragma HLS INTERFACE ap_ctrl_none  port=return
27     // ...
28 }

```

Listing 5.4: Schnittstellendefinition der HLS-Module (Ausschnitt)

Beide Module sind vollständig streamingfähig und verwenden eine `ap_ctrl_none`-Schnittstelle. Dadurch erfolgt die Verarbeitung automatisch beim Eintreffen gültiger Eingangsdaten – ein explizites Startsignal ist nicht erforderlich. Der gesamte Datenpfad ist vollständig datengetrieben und reagiert ausschließlich auf den Takt und das Vorhandensein von Daten [47].

Zur Laufzeitüberwachung wurde im Modul `ofdm_tx` zusätzlich ein Steuerport **led** implementiert, der – wie bereits in Abschnitt 5.4.1 beschrieben – den aktiven Verarbeitungsstatus visuell signalisiert. Diese einfache Zustandsanzeige unterstützt die manuelle Verifikation während der Entwicklung und des Testbetriebs. Sämtliche Module arbeiten innerhalb der **FPGA_CLK**-Domäne und sind strikt taktsynchron aufgebaut.

5.5.2 Signalverarbeitungsblöcke (16QAM, IFFT, CP)

Die FPGA-basierte Verarbeitungskette folgt der in Abschnitt 4.7 spezifizierten Signalflussarchitektur. Die zentrale OFDM-Verarbeitung gliedert sich in drei logisch getrennte Funktionsblöcke:

- Modulation/Demodulation mittels 16-QAM,
- Transformationen über IFFT bzw. FFT,
- Hinzufügen und Entfernen des zyklischen Präfixes.

Jeder dieser Verarbeitungsschritte wurde als separate Funktionseinheit in HLS realisiert und im Rahmen der Gesamtintegration schrittweise eingebunden. Die funktionale Verifikation der einzelnen Module erfolgte iterativ: Für jede Stufe wurden mit dem MATLAB-Referenzmodell gezielt definierte Testvektoren erzeugt und in einer HLS-Testbench evaluiert.

Dabei wurde überprüft, ob die Ausgaben der jeweiligen HLS-Funktionen exakt mit den Ergebnissen des MATLAB-Modells übereinstimmen. Diese Validierungsstrategie ermöglichte eine schrittweise Integration unter Wahrung der Funktionssicherheit und gewährleistete die Konsistenz zwischen dem Hardware-Modell und dem mathematischen Referenzmodell.

1) Mapping / Demapping

Die Modulations- und Demodulationsstufen sind vollständig in den HLS-Modulen für Sender (TX) und Empfänger (RX) getrennt implementiert. Beide Blöcke basieren auf dem in Abschnitt 2.1.3 beschriebenen 16QAM-Schema. Die Umsetzung erfolgt nach einem expliziten **Dataflow**-Ansatz mit angewendeter *Pipelining*-Optimierung, um eine maximale Parallelität bei minimaler Verarbeitungslatenz zu erreichen [62].

Mapping (Senderseite) Jedes über die AXI4-Stream-Schnittstelle eintreffende 16-Bit-Datenwort enthält vier 16QAM-Symbole [17]. Das Datenwort wird in vier Nibbles zu je 4 Bit unterteilt, wobei die Reihenfolge der Bitgruppen erhalten bleibt:

- $n0 = \text{Bits}[3:0]$
- $n1 = \text{Bits}[7:4]$
- $n2 = \text{Bits}[11:8]$
- $n3 = \text{Bits}[15:12]$

Zur Erzeugung des zugehörigen komplexen Symbols werden die oberen beiden Bits (Bit 3:2) als Realteil und die unteren beiden Bits (Bit 1:0) als Imaginärteil interpretiert. Die Bitmuster werden dabei gemäß einer Gray-Codierung auf diskrete Amplituden abgebildet:

$$00 \rightarrow -3, \quad 01 \rightarrow -1, \quad 10 \rightarrow +1, \quad 11 \rightarrow +3$$

Diese Abbildung erfolgt sowohl für den Real- als auch für den Imaginärteil, sodass sich pro Nibble genau ein komplexer Konstellationspunkt ergibt.

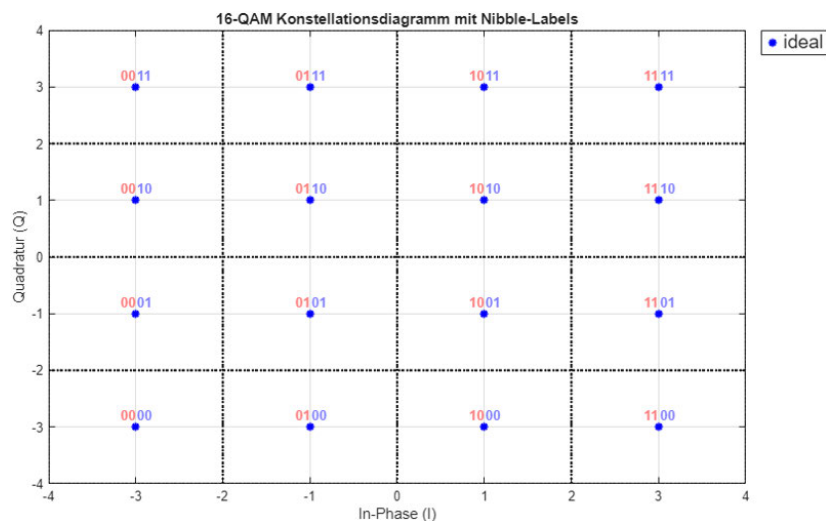


Abbildung 5.11: Zuordnung der 16 möglichen 4-Bit-Nibbles zu den 16-QAM Konstellationspunkten (Realteil-Bits) und (Imaginärteil-Bits)

Zum Beispiel bei einer 16-Bit-Zahl $0xABCD$ ergibt sich folgendes Bitmuster:

$$(0xABCD)_{16} = (1010 \ 1011 \ 1100 \ 1101)_2$$

Die einzelnen 4-Bit-Nibbles lauten entsprechend:

$$n0 = 0xA = 1010, \quad n1 = 0xB = 1011, \quad n2 = 0xC = 1100, \quad n3 = 0xD = 1101$$

Unter Anwendung des oben beschriebenen Mappings ergeben sich die folgenden komplexen Symbole:

- $n0 = (1010)_2 \Rightarrow \text{Bit } 3:2 = 10 \Rightarrow \text{Re} = +1, \quad \text{Bit } 1:0 = 10 \Rightarrow \text{Im} = +1 \Rightarrow z_0 = 1 + 1i$
- $n1 = (1011)_2 \Rightarrow \text{Bit } 3:2 = 10 \Rightarrow \text{Re} = +1, \quad \text{Bit } 1:0 = 11 \Rightarrow \text{Im} = +3 \Rightarrow z_1 = 1 + 3i$
- $n2 = (1100)_2 \Rightarrow \text{Bit } 3:2 = 11 \Rightarrow \text{Re} = +3, \quad \text{Bit } 1:0 = 00 \Rightarrow \text{Im} = -3 \Rightarrow z_2 = 3 - 3i$
- $n3 = (1101)_2 \Rightarrow \text{Bit } 3:2 = 11 \Rightarrow \text{Re} = +3, \quad \text{Bit } 1:0 = 01 \Rightarrow \text{Im} = -1 \Rightarrow z_3 = 3 - 1i$

Diese symbolische Darstellung ermöglicht eine effiziente Parallelisierung der Modulationslogik und vereinfacht die Implementierung in HLS. Die resultierenden Punkte liegen auf einem regulären 4×4 -Gitter im komplexen Raum, wie in Abbildung 5.11 dargestellt.

Demapping (Empfängerseite) Die Demodulation im RX-Modul erfolgt in umgekehrter Reihenfolge. Die aus der FFT resultierenden komplexwertigen Samples im Q1.15-Fixpunktformat werden auf die jeweils nächstgelegenen Konstellationspunkte der 16QAM-Menge abgebildet. Dabei erfolgt eine Schwellenwertentscheidung basierend auf der normierten Amplitudenskala $\{-3, -1, +1, +3\}$ (normiert auf $\sqrt{10}$).

Anschließend werden Real- und Imaginärteil getrennt betrachtet und jeweils wieder in zwei Bits codiert. Aus jedem komplexen Symbol entsteht somit erneut ein 4-Bit-Nibble. Vier dieser Nibbles werden anschließend wieder zu einem 16-Bit-Datenwort zusammengeführt – exakt in derjenigen Bitreihenfolge, wie sie zuvor auf der Senderseite erzeugt wurde. Dadurch ist eine verlustfreie Rekonstruktion der ursprünglichen Datenstruktur gewährleistet.

2) Bildung des Frequenzrasters, Einfügung von Piloten und Schutzintervall sowie Subträgerextraktion

Wie in Kapitel 4.2 erläutert, wurden die Anzahl der Datensubträger, der Pilotsymbole sowie die Dauer des Schutzintervalls (Cyclic Prefix) so gewählt, dass für unterschiedliche FFT-Längen stets dieselbe effektive Datenrate erzielt wird. Diese Randbedingungen bilden die Grundlage für die Subträgerbelegung im Frequenzbereich sowohl auf der Sender- als auch auf der Empfängerseite.

Ausgehend von den in Tabelle 4.1 definierten Systemparametern wurde eine MATLAB-Funktion implementiert, welche für jede betrachtete FFT-Länge die Indizes der Datensubträger, der Piloten sowie der Nullträger (DC-Träger, Nyquist-Träger und Guard-Bänder) automatisch berechnet und als Index-Arrays ausgibt. Diese Index-Arrays werden anschließend sowohl im Sender als auch im Empfänger verwendet, um eine konsistente Positionierung der Datensubträger über das gesamte OFDM-System hinweg sicherzustellen.

Als Beispiel ist in Listing 5.5 die Subträgerbelegung für den Fall $N = 128$ dargestellt.

```
1  static const int STAGES      = 7;          /* Stufen der FFT */
2  static const int N_FFT      = 128;         /* FFT-Länge */
3  static const int N_CP       = N_FFT/8;     /* CP-Länge */
4  static const int N_FFT_HALF = N_FFT/2;     /* Hälfte der FFT-Länge */
5  static const int NUM_PIL    = 8;           /* Anzahl der Pilotsubträger */
6  static const int NUM_NULL   = 16;          /* Anzahl der DC + Nyquist-Träger + Guard-
   Band */
7  static const int NUM_DATA   = 48;          /* Anzahl der Datensubträger */
8
9  /* Piloten (positive Seite: 1..63) */
10 static const int PILOTS[NUM_PIL] = {7, 14, 21, 28, 35, 42, 49, 56};
11
12 /* Daten (positive Seite: 1..63) */
13 static const int DATA_K[NUM_DATA] = {
14     4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 29, 30, 31,
15     32, 33, 34, 36, 37, 38, 39, 40, 41, 43, 44, 45, 46, 47, 48, 50, 51, 52, 53, 54, 55, 57, 58, 59};
16
17 /* DC-Position + Nyquist-Träger + Guard-Band */
18 static const int NULL_IDX[NUM_NULL] =
   {0, 1, 2, 3, 60, 61, 62, 63, 64, 65, 66, 67, 68, 125, 126, 127};
```

Listing 5.5: Subträgerindizes (Daten, Piloten, Nullträger) für OFDM-System mit $N = 128$

Frequenzraster und Subträgerbelegung im Sender Nach der QAM-Modulation werden die komplexen Datensymbole entsprechend der berechneten Index-Arrays in das Frequenzraster der IFFT eingebettet. Dabei werden

- die Datensubträger ausschließlich auf den vorgesehenen positiven Frequenzpositionen platziert,
- die Pilotsymbole an fest definierten, äquidistant verteilten Subträgerpositionen eingefügt,
- die DC-Position, der Nyquist-Träger sowie die äußeren Guard-Bänder als Nullträger belegt.

Zur Erzeugung eines reellwertigen Zeitsignals wird für die negative Frequenzhälfte das hermitesch symmetrische Spektrum gebildet, indem die komplex konjugierten Werte der positiven Frequenzhälfte gespiegelt werden. Das resultierende vollständige Frequenzraster bildet anschließend das Eingangsarray der IFFT.

Einfügung des Schutzintervalls Nach der IFFT werden die resultierenden Zeitbereichssignale um ein Schutzintervall erweitert. Hierzu werden die letzten N_{CP} Abtastwerte eines OFDM-Symbols entnommen und als Cyclic Prefix am Anfang des OFDM-Frames eingefügt. Das Schutzintervall dient der Unterdrückung von Inter-Symbol-Interferenz sowie der Erhöhung der Robustheit gegenüber Mehrwegeausbreitung im Übertragungskanal [18].

Subträgerextraktion im Empfänger Auf der Empfängerseite wird zunächst das Schutzintervall gemäß der bekannten CP-Länge entfernt. Anschließend erfolgt die FFT zur Rücktransformation in den Frequenzbereich. Unter Verwendung derselben Index-Arrays wie im Sender werden danach ausschließlich die Datensubträger aus dem FFT-Spektrum extrahiert, während Piloten, Nullträger sowie die negative Frequenzhälfte verworfen werden.

Die extrahierten Datensubträger werden anschließend der QAM-Demodulation zugeführt.

3) FFT- und IFFT-Verarbeitungseinheit

Wie bereits in Abschnitt 4.3 beschrieben, basiert die Umsetzung von IFFT und FFT auf dem Radix-2-Algorithmus, der speziell für eine effiziente Implementierung in High Level Synthesis (HLS) optimiert wurde [6]. Dabei wurden die zugehörigen Twiddle-Faktoren als statisches Array abgelegt. Nach der funktionalen Verifikation mittels Testbench erfolgte die Synthese des Moduls. Dabei zeigte sich jedoch, dass die Ressourcennutzung insbesondere in Form von Look-Up-Tables (LUTs) und Flip-Flops (FFs) sehr hoch ausfiel und die maximal erreichbare Taktfrequenz deutlich unter den Anforderungen lag. Aus diesem Grund wurde die Architektur anschließend gezielt durch den Einsatz von HLS-Pragmas optimiert.

Speicherorganisation und Datenhaltung Die Zwischenergebnisse der einzelnen FFT-Stufen werden in einem sogenannten Ping-Pong-Buffer gespeichert [36]. Hierzu werden zwei Speicherblöcke verwendet, die abwechselnd als Ein- und Ausgabepuffer dienen. Durch diese Struktur können Lese- und Schreibzugriffe parallel erfolgen, was die effektive Latenz der Berechnung deutlich reduziert. Beide Speicher wurden explizit als Dual-Port-BRAM (BRAM_2P) instanziiert:

```
cpx_tw bufA[N_FFT]
#pragma HLS RESOURCE variable=bufA core=RAM_2P
cpx_tw bufB[N_FFT]
#pragma HLS RESOURCE variable=bufB core=RAM_2P
```

Die Twiddle-Faktoren werden in einem separaten BROM gespeichert. Durch vollständige Array-Partitionierung stehen alle Koeffizienten ohne zusätzliche Zugriffswartezeiten parallel zur Verfügung:

```
extern const cpx_tw TW[N_FFT_HALF];
#pragma HLS RESOURCE variable=TW core=ROM_1P
#pragma HLS ARRAY_PARTITION variable=TW complete dim=1
```

Pipeline-Struktur der FFT-Stufen Bei einer 128-Punkte-IFFT wird in sieben aufeinanderfolgenden Stufen nach dem Decimation-in-Frequency-Verfahren (DIF) berechnet [24]. Jede Stufe besteht aus exakt 64 Butterfly-Operationen. Um eine maximale Parallelität

zu erreichen, wurden die Schleifen vollständig „geflattet“, sodass pro Stufe genau eine Butterfly-Schleife existiert.

Die Implementierung der inneren Butterfly-Schleife der IFFT ist in Vitis HLS mit einem festen Pipeline-Intervall von $II = 3$ versehen. Dies bedeutet, dass alle drei Taktzyklen eine neue Iteration gestartet werden kann, wodurch eine kontinuierliche Pipeline-Verarbeitung mehrerer Butterfly-Operationen ermöglicht wird. Die für jede Butterfly-Stufe erforderlichen komplexen Multiplikationen mit den Twiddle-Faktoren werden explizit auf DSP-Blöcke des FPGA abgebildet, um die Rechenleistung optimal auszunutzen. Darüber hinaus treten bei klassischen Ping-Pong-Pufferstrukturen in HLS häufig automatisch erkannte Speicherabhängigkeiten auf, da Quell- und Zielpuffer innerhalb der Schleifenstruktur ineinandergreifen können [61]. In der Implementierung wird dieses Problem durch eine strikte Trennung von Lese- und Schreibpuffern in jeder Stufe gelöst. Durch diese explizite Trennung kann der Compiler alle speicherbezogenen Abhängigkeiten aufheben. In der Folge ist es dem HLS-Synthesetool möglich, die innere Schleife vollständig zu pipelinen, sodass mehrere Butterfly-Operationen gleichzeitig im Datenpfad verarbeitet werden können. Dies verbessert die Gesamtlatenz der IFFT erheblich und erhöht den Durchsatz der Datenverarbeitung.

```
#pragma HLS PIPELINE II=3
#pragma HLS ALLOCATION instances=cmul limit=8 operation
#pragma HLS BIND_OP op=mul impl=DSP
```

Durch diese Optimierungen konnte sowohl die Rechenzeit der IFFT deutlich reduziert als auch die maximale Taktrate der resultierenden Hardware signifikant erhöht werden.

Normierung durch Schiebeoperationen Die Normierung der IFFT erfolgt durch eine Division um den Faktor 2 in jeder der sieben Stufen, was insgesamt einer Gesamtnormierung von $1/128$ entspricht. Anstelle einer Fixpunkt-Multiplikation mit dem Faktor 0.5 wird diese Skalierung durch eine arithmetische Rechtsverschiebung ($\gg 1$) realisiert.

Die Verwendung von Schiebeoperationen bietet zwei Vorteile. Erstens erfolgt die Skalierung bitgenau ohne Rundungsfehler, da lediglich eine Verschiebung der Bits vorgenommen wird und weder Rundung noch Sättigung auftritt. Zweitens wird der Einsatz zusätzlicher DSP-Ressourcen vermieden, da eine Multiplikation mit 0.5 in der Hardware stets als echte Fixpunkt-Multiplikation umgesetzt wird, die ein breiteres Zwischenergebnis erzeugt und anschließend gerundet sowie saturiert werden muss.

Praktische Experimente zeigten, dass bei einer Normierung mittels Multiplikation insbesondere bei reduzierter Bitbreite der Twiddle-Faktoren ein systematisches Drift-Verhalten im gesamten OFDM-TX-/RX-System auftritt. Die Ursache hierfür sind die sich über alle Butterfly-Stufen und OFDM-Symbole hinweg akkumulierten Rundungsfehler. Durch die Umstellung auf Schiebeoperationen konnte dieser Effekt vollständig eliminiert werden.

Bit-Reversal und Ausgabe Nach Abschluss der sieben Rechenstufen liegen die Zeitbereichswerte in bit-invertierter Reihenfolge vor. Diese werden in einem separaten Schritt durch eine Bit-Reverse-Adressierung wieder in die natürliche Reihenfolge überführt. Gleichzeitig erfolgt die Rückquantisierung von der internen Fixpunktdarstellung auf das externe Q1.15-Format für die Weiterverarbeitung im OFDM-Sender.

Durch den gezielten Einsatz von Dual-Port-Ping-Pong-Speichern, einer vollständig partitionierten Twiddle-ROM-Struktur, konsequenter Pipeline-Optimierung der Butterfly-Operationen, DSP-basierter komplexer Multiplikation sowie einer normierenden Skalierung mittels Schiebeoperationen konnte eine rechenzeitoptimierte und ressourceneffiziente FFT-/IFFT-Architektur realisiert werden.

Parametrisierung und Validierung Nach der Optimierung wurde das FFT-/IFFT-Modul verifiziert und mit einem Referenzmodell in Matlab verglichen. Dabei wurden sowohl der Zeit- als auch der Frequenzbereich sowie die numerische Genauigkeit der Ergebnisse untersucht. Auf Basis dieser Validierung wurden zentrale Einstellparameter wie die Pipeline-Stufentiefe, die IFFT-/FFT-Länge sowie die Anzahl der verwendeten Twiddle-Faktoren als globale Parameter im Design deklariert (vgl. Listing 5.5).

Durch diese Parametrisierung ist es möglich, unterschiedliche FFT-Längen flexibel zu konfigurieren. In der vorliegenden Arbeit werden insbesondere die drei IFFT-/FFT-Längen

$$N \in \{64, 128, 256\}$$

systematisch untersucht. Dadurch kann der Einfluss der FFT-Länge auf Rechenzeit, Ressourcenauslastung und numerische Stabilität des OFDM-Systems gezielt analysiert werden.

5.5.3 HLS-basierte Ressourcenschätzung für 16-Bit-Twiddle-Faktoren

In diesem Abschnitt werden die von Vitis HLS ermittelten Ressourcenschätzungen für die FFT-/IFFT-Implementierung dargestellt. Die angegebenen Werte basieren auf der HLS-Synthese vor dem eigentlichen RTL-Synthese- und Implementierungsschritt und stellen daher eine vorläufige Abschätzung dar. Die Auswertung erfolgt für die drei betrachteten FFT-Längen $N \in \{64, 128, 256\}$ bei einer internen Twiddle-Faktor-Breite von 16 Bit und Taktfrequenz von 100 MHz. Die angegebenen Ressourcenwerte dienen primär dem relativen Vergleich der verschiedenen FFT-Konfigurationen und stellen keine finalen Implementierungsergebnisse auf dem Ziel-FPGA dar.

Resource	FFT 64	FFT 128	FFT 256
LUT	3800	4872	6598
LUTRAM	0	0	0
FF	3242	4092	5074
BRAM	10	10	10
DSP	4	4	4

Tabelle 5.3: Vitis HLS – Ressourcenschätzung für FFT-Längen $N \in \{64, 128, 256\}$ bei 16-Bit-Twiddle-Faktoren (Sender)

Resource	FFT 64	FFT 128	FFT 256
LUT	3231	3509	4345
LUTRAM	0	0	0
FF	1509	1529	1552
BRAM	8	8	8
DSP	4	4	4

Tabelle 5.4: Vitis HLS – Ressourcenschätzung für FFT-Längen $N \in \{64, 128, 256\}$ bei 16-Bit-Twiddle-Faktoren (Empfänger)

Geschätzte Latenz in Vitis HLS Neben der Ressourcennutzung wurde in Vitis HLS auch die Latenz der OFDM-Sende- und -Empfangskette für die unterschiedlichen FFT-Längen abgeschätzt. Die in Tabelle 5.5 aufgeführten Werte geben die Gesamtlatenz in Taktzyklen für die Datenpfade mit einer Twiddle-Faktor-Breite von 16 Bit und Taktfrequenz von 100 MHz an.

Pfad	FFT 64	FFT 128	FFT 256
Sender	732	1628	3614
Empfänger	725	1621	3605

Tabelle 5.5: Vitis HLS – Latenzschätzung in Taktzyklen für die OFDM-Sende- und -Empfangspfade bei FFT-Längen $N \in \{64, 128, 256\}$ und 16-Bit-Twiddle-Faktoren

5.6 Sender- und Empfängerseitige Verarbeitung

5.6.1 Senderkette (TX)

Die vollständige zeitliche Verarbeitungsabfolge der Senderkette ist im elektronischen Anhang in Abbildung *Verarbeitungssequenz_TX.png* dargestellt. Beginnend bei der Datenbereitstellung vom DSP werden die Nutzdaten über das EMIFA-Interface in den Eingangsspeicher des Systems übertragen und anschließend durch den HLS-basierten OFDM-Sender verarbeitet. Die einzelnen Sequenzen sind nummeriert und zeigen schrittweise den internen Datenfluss bis zur Ablage des generierten OFDM-Zeitsignals im Ausgangs-BRAM, dann kann der vollständige OFDM-Frame anschließend vom DSP ausgelesen werden.

Wie bereits in Abschnitt 5.4.1 beschrieben, wurde im **EMIFA_Interface** ein Taktzähler für die jeweiligen Verarbeitungspfade implementiert. Damit kann für den Sendepfad die benötigte Verarbeitungszeit gemessen werden – beginnend mit dem Auslesen der **FIFO_IN** im FPGA, also unmittelbar nach dem Setzen des **GO**-Bits, bis hin zum Setzen des **DONE**-Bits auf FPGA-Seite.

In mehreren Testdurchläufen mit $N = 128$ einer internen Twiddle-Faktor-Breite von 16 Bit sowie einer Betriebstaktfrequenz von 100 MHz, wurden konstant **1873** Taktzyklen

für die gesamte Verarbeitung gemessen. Dieser Wert wurde über tausende Iterationen stabil reproduziert.

$$t_{\text{Verarbeitung}} = \frac{1873}{100 \text{ MHz}} = 18.73 \mu\text{s}$$

Diese Zeit enthält ausschließlich die interne Verarbeitung im FPGA und berücksichtigt noch nicht die Zugriffszeiten über den EMIFA-Bus. Dazu kommen gemäß Gleichung 5.4 noch die Buszugriffszeiten mittels EDMA-Controller über EMIFA – sowohl für das Schreiben von 12 Datenwörtern als auch für das Lesen des kompletten OFDM-Frames mit 144 Werten inklusive Cyclic Prefix:

$$t_S = 12 \cdot 60 \text{ ns} + (12 \bmod 8) \cdot 20 \text{ ns} = 0.74 \mu\text{s}$$

$$t_L = 144 \cdot 60 \text{ ns} + (144 \bmod 8) \cdot 80 \text{ ns} = 10.08 \mu\text{s}$$

Daraus ergibt sich eine gesamte EMIFA-Zugriffszeit von:

$$t_{\text{EMIFA}} = t_S + t_L = 10.82 \mu\text{s}$$

Somit ergibt sich für den gesamten Sendepfad ein Zeitfenster von:

$$t_{\text{sender}} = t_{\text{Verarbeitung}} + t_{\text{EMIFA}} = 18.73 \mu\text{s} + 10.82 \mu\text{s} = 29.55 \mu\text{s}$$

5.6.2 Empfängerkette (RX)

Die zeitliche Verarbeitungsabfolge der Empfängerseite ist entsprechend im elektronischen Anhang in Abbildung *Verarbeitungssequenz_RX.png* dargestellt. Die empfangenen Zeitbereichsdaten werden vom DSP über das EMIFA-Interface in den Eingangsspeicher geschrieben und anschließend an den HLS-basierten OFDM-Empfänger übergeben. Die nummerierten Sequenzen zeigen den synchronisierten Ablauf vom Startsignal über FFT und Demodulation bis zur Speicherung der dekodierten Nutzdaten im Ausgangs-BRAM. Die zurückgewonnenen Daten können anschließend wieder durch den DSP ausgelesen werden.

Für die empfangsseitige Verarbeitung wurde der Taktzähler zur Messung der Verarbeitungszeit im FPGA eingesetzt. Die gemessene Zyklusanzahl beträgt für eine FFT-Länge

von $N = 128$ mit 16-Bit-Twiddle-Faktoren konstant:

$$t_{\text{Verarbeitung}} = \frac{1829}{100 \text{ MHz}} = 18.29 \mu\text{s}$$

Diese Zeit berücksichtigt ausschließlich die interne Verarbeitung im FPGA und schließt die Zugriffszeiten über den EMIFA-Bus noch nicht ein.

Gemäß Gleichung 5.4 ergeben sich zusätzlich die Buszugriffszeiten mittels EDMA-Controller über EMIFA – sowohl für das Schreiben des kompletten OFDM-Frames mit 144 Werten als auch für das Lesen von 12 Datenwörtern:

$$t_S = 144 \cdot 60 \text{ ns} + (144 \bmod 8) \cdot 20 \text{ ns} = 9 \mu\text{s}$$

$$t_L = 12 \cdot 60 \text{ ns} + (12 \bmod 8) \cdot 80 \text{ ns} = 0.8 \mu\text{s}$$

Daraus ergibt sich eine gesamte EMIFA-Zugriffszeit von:

$$t_{\text{EMIFA}} = t_S + t_L = 9.80 \mu\text{s}$$

Somit ergibt sich für den gesamten Empfangspfad ein Zeitfenster von:

$$t_{\text{empfänger}} = t_{\text{Verarbeitung}} + t_{\text{EMIFA}} = 18.29 \mu\text{s} + 9.80 \mu\text{s} = 28.09 \mu\text{s}$$

Gesamtlatenz Sender- und Empfängerketten

Die zuvor analysierten Verarbeitungszeiten für Sender- und Empfängerketten lassen sich zu einer Gesamtlatenz zusammenfassen. Dabei werden sowohl die reinen Verarbeitungszeiten im FPGA (gemessen über den integrierten Taktzähler) als auch die Buszugriffszeiten über EMIFA mittels EDMA-Controller berücksichtigt. Für den betrachteten Fall mit einer FFT-Länge von $N = 128$, einer internen Twiddle-Faktor-Breite von 16 Bit und einer Betriebsfrequenz von 100 MHz ergeben sich folgende Zeitanteile:

- **Senderpfad:** $t_{\text{sender}} = 29.55 \mu\text{s}$
- **Empfängerpfad:** $t_{\text{empfänger}} = 28.09 \mu\text{s}$

Somit ergibt sich für den vollständigen Übertragungspfad eine gesamte Nettolatenz von:

$$T = t_{\text{sender}} + t_{\text{empfänger}} = 29,55 \mu\text{s} + 28,09 \mu\text{s} = 57.64 \mu\text{s}$$

Diese Gesamtdauer basiert auf der Taktzyklusmessung innerhalb des FPGAs sowie den berechneten EDMA-Zugriffszeiten und stellt somit die Nettolatenz für eine vollständige blockbasierte OFDM-Übertragung dar – bestehend aus Datenbereitstellung, Verarbeitung, Frame-Generierung sowie dem Zurücklesen der dekodierten Nutzdaten. Im Kapitel 6 werden diese Zeiten im Kontext des Echtzeitbetriebs weiter analysiert und bewertet.

5.7 Vergleichsbasis DSP-Only

Für eine faire und objektive Bewertung der Performanz wurde das OFDM-System in einer rein DSP-basierten Variante implementiert und untersucht. Dabei wurde sichergestellt, dass in beiden Implementierungsvarianten dieselben algorithmischen Verarbeitungsansätze zum Einsatz kommen. Die grundlegenden Verarbeitungsschritte (IFFT/FFT, Modulation/Demodulation, Normierung sowie Frequenzraster und Subträgerbelegung), die ursprünglich hardwarebeschleunigt in HLS für die FPGA-Implementierung entworfen wurden, wurden entsprechend in C-Code übertragen und so optimiert, dass sie auf dem DSP ausführbar sind [36] [24].

Um die Vergleichbarkeit sicherzustellen, wurden identische Parameter verwendet – insbesondere:

- FFT-/IFFT-Länge: $N \in \{64, 128, 256\}$
- Twiddle-Faktor-Breite: 16 Bit Fixpunkt
- Modulationsschema: 16QAM
- identische Subträgerbelegung und CP-Länge

Auf diese Weise kann eine sachlich fundierte Leistungsanalyse beider Architekturen durchgeführt werden, bei der die reinen Rechenzeiten und Ressourcenanforderungen unter vergleichbaren Bedingungen gegenübergestellt werden.

5.8 Vorwärtsfehlerkorrektur (FEC)

5.8.1 Faltungscode: Encoder und Decoder

Wie in Abschnitt 2.2 erläutert, wird der Faltungscodex in dieser Arbeit ausschließlich zur Überprüfung der Erweiterbarkeit der entwickelten Sender- und Empfängermodule in HLS eingesetzt. Der Fokus liegt dabei nicht auf einer hochoptimierten FEC-Implementierung, sondern auf der funktionalen Integration in die bestehende OFDM-Verarbeitungskette. Der Encoder und der Decoder wurden daher lediglich so weit optimiert, dass ein stabiler Betrieb bei einer minimalen Taktfrequenz von 50 MHz möglich ist und die verfügbaren FPGA-Ressourcen nicht überschritten werden [58] [37].

Für die Implementierung wurde ein frei verfügbarer Open-Source-Faltungsgcode mit Viterbi-Decoder verwendet und für den Einsatz in Vitis HLS angepasst [43]. Die Optimierung beschränkte sich dabei im Wesentlichen auf die Einbindung geeigneter HLS-Pragmas zur Sicherstellung der Synthesefähigkeit und der geforderten Taktrate. Eine weitergehende architekturelle Optimierung des Decoders wurde bewusst nicht vorgenommen, da dieser lediglich als Funktionsnachweis für die Erweiterbarkeit des Gesamtsystems dient.

Der verwendete Faltungscode ist ein $(2, 1, 6)$ -Code mit den Generatorpolynomen

$$G_1(X) = 1 + X^2 + X^3 + X^5 + X^6 \quad \Rightarrow \quad g_1[n] = \{1, 0, 1, 1, 0, 1, 1\} \quad (5.9)$$

$$G_2(X) = 1 + X + X^2 + X^3 + X^6 \Rightarrow g_2[n] = \{1, 1, 1, 1, 0, 0, 1\} \quad (5.10)$$

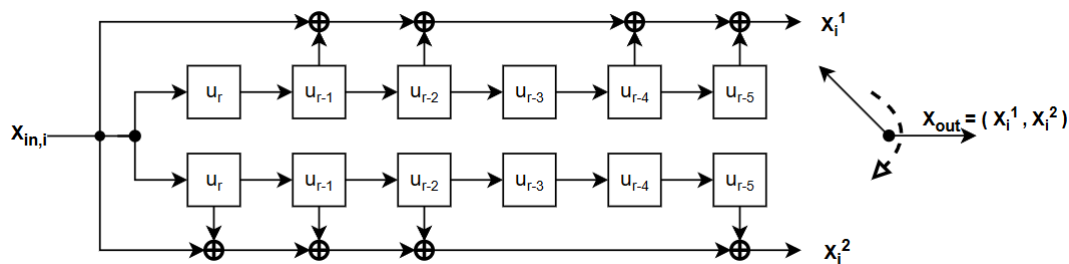


Abbildung 5.12: Encoder eines $(2, 1, 6)$ -Faltungscodes in Schieberegister-Darstellung

Die Encoderstruktur basiert auf einem Schieberegister mit sieben Speicherstellen, wobei pro Eingangsbit zwei codierte Ausgangsbits erzeugt werden. Abbildung 5.12 zeigt die Schieberegister-Darstellung des verwendeten Encoders.

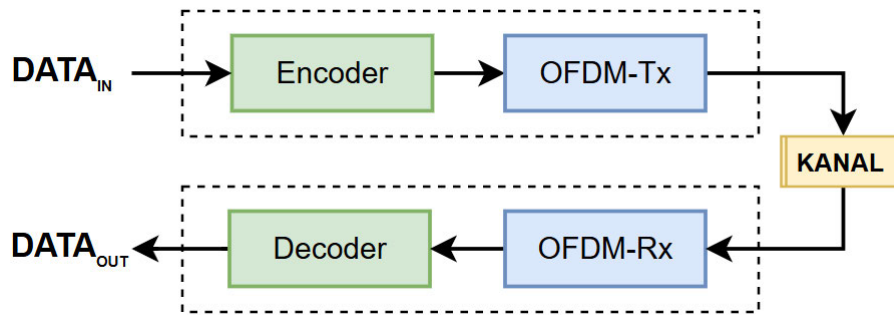


Abbildung 5.13: Schematische Integration des FEC-Encoders und -Decoders in die OFDM-Verarbeitungskette

Auf der Senderseite wurde der FEC-Encoder vor dem *ofdm-tx*-IP-Core eingefügt, sodass die zu übertragenden Daten vor der Modulation mit redundanten Informationen angereichert werden [59] [32]. Auf der Empfängerseite wurde der FEC-Decoder direkt hinter dem *ofdm-rx*-IP-Core positioniert, um die empfangenen Daten nach der Demodulation zu dekodieren und rekonstruieren. Abbildung 5.13 zeigt die schematische Einbindung der FEC-Komponenten in die Datenpfade.

5.9 Verifikation und Co-Simulation

Nach der erfolgreichen Verifikation der einzelnen Module durch Vergleich mit entsprechenden Referenzmodellen in MATLAB folgt in diesem Abschnitt die Integration aller Komponenten zu einem Gesamtsystem. Ziel ist es, das vollständig System unter realistischen Bedingungen zu testen und dessen Funktionalität sicherzustellen.

5.9.1 Gesamtsystemverifikation

Das verwendete OFDM-System basiert auf folgenden Parametern:

Ausgehend von 48 Datensubträgern mit je einem modulierten 16QAM-Symbol (4 Bit pro Symbol) ergibt sich ein Datenumfang von $48 \times 4 = 192$ Bit pro OFDM-Symbol. Da der DSP intern mit 16-Bit-Datenwörtern arbeitet, sind insgesamt $192/16 = 12$ Datenwörter erforderlich, um ein komplettes OFDM-Symbol zu bilden.

Symbol	Bedeutung
Modulationsart	16QAM
N	128 (IFFT/FFT-Länge)
N_{Daten}	48 (Anzahl Datensubträger)
N_{Pilot}	16 (Anzahl Pilotsubträger)
N_{CP}	16 (Cyclic Prefix)
F_s	100 kHz (Abtastrate ADC/DAC)
Δf	781.25 Hz (Subträgerabstand)
T_u	1.28 ms (Nutzsymbolsymbol ohne CP)
T_{CP}	0.16 ms (Schutzintervall)
T_{sym}	1.44 ms (Gesamtsymbol inkl. CP)
F_{FPGA}	100 MHz (FPGA CLK)
F_{EMIFA}	100 MHz (EMIFA CLK)

Tabelle 5.6: Systemparameter des verwendeten OFDM-Systems

Test 1: TX-Verarbeitung und Matlab-Demodulation Im ersten Verifikationstest werden bekannte 12 Datenwörter als Eingabe in den TX-Pfad gegeben. Das resultierende OFDM-Zeitsignal wird im Ausgangs-FIFO gespeichert, vom DSP ausgelesen und anschließend im MATLAB-Modell demoduliert. Die zurückgewonnenen Daten stimmen bitgenau mit den ursprünglichen Daten überein.

Test 2: Matlab-Signal und RX-Pfad Im zweiten Test wird ein synthetisch erzeugter OFDM-Frame direkt im MATLAB-Modell generiert, über den EMIFA-Bus in den FPGA geschrieben und anschließend im RX-Pfad verarbeitet. Auch hier konnten die ursprünglichen Daten korrekt rekonstruiert werden, was die fehlerfreie Funktion des RX-Moduls bestätigt.

Systemintegration und Live-Test Nachdem die einzelnen Module erfolgreich getestet wurden, erfolgte die vollständige Systemintegration und Verbindung beider Pfade. In einem Echtzeit-Szenario wurde ein analoges Audiosignal mit 100 kHz am ADC abgetastet, in 12er-Blöcken gesammelt und durch das OFDM-System (TX- und RX-Pfad) verarbeitet. Die dekodierten Signale wurden anschließend über den DAC wieder ausgegeben.

Die zugrunde liegende Verarbeitung erfolgt pipelining-basiert (vgl. Abbildung 6.1): Während ein Datenblock verarbeitet wird, wird bereits der nächste vorbereitet. Die Verarbeitungszeit ergibt sich aus:

- Abtastzeit für 12 Samples: $12/100 \text{ kHz} = 120 \mu\text{s}$
- Übertragung und TX-Verarbeitung (inkl. EMIFA): $\approx 29.55 \mu\text{s}$
- RX-Verarbeitung (inkl. EMIFA): $\approx 28.09 \mu\text{s}$

Die Gesamtverarbeitungskette (Sammlung, Übertragung, OFDM-Verarbeitung und Ausgabe) wurde kontinuierlich ausgeführt. Die vollständige Systemstruktur ist in Abbildung 5.14 dargestellt.

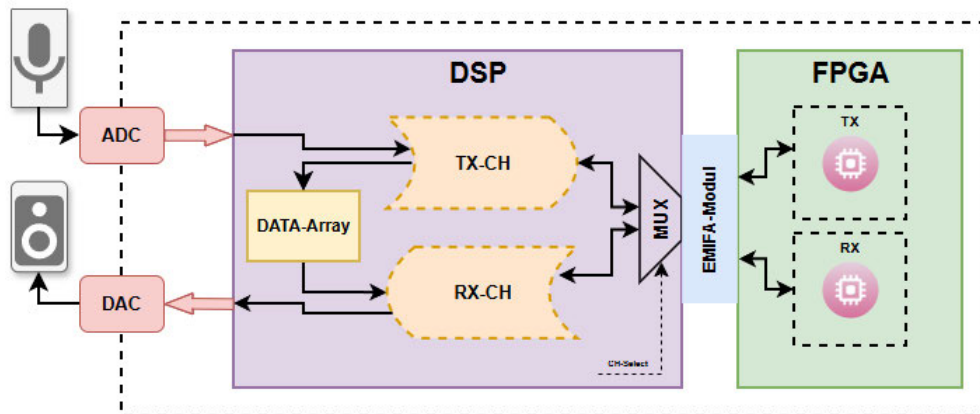


Abbildung 5.14: Systemaufbau zur Gesamtsystemverifikation (DSP/FPGA)

Die zeitliche Analyse der vollständigen Verarbeitungstrecke wurde mithilfe des **Code Composer Studio (CCS)** durchgeführt. Anhand der gemessenen DSP-Zyklen (bei 456 MHz Taktfrequenz) wurde die Verarbeitungsdauer der Buszugriffe und Steueralgorithmus zusätzlich bestimmt. Die detaillierte Analyse folgt in Kapitel 6.

6 Ergebnisse

Dieses Kapitel dokumentiert die finalen Ergebnisse der Implementierung und Verifikation des OFDM-Systems. Im Fokus steht die funktionale Integrität der gesamten Signalverarbeitungskette, bestehend aus Sender, Empfänger und optionalen Erweiterungen wie Vorwärtsfehlerkorrektur (FEC). Neben der Verifikation auf Modulebene wurden auch Systemtests durchgeführt, die die Einhaltung der in Kapitel 3 definierten funktionalen und nicht-funktionalen Anforderungen belegen. Darüber hinaus werden zentrale Erkenntnisse und Herausforderungen, die während der Implementierungsphase identifiziert wurden, zusammenfassend dargestellt.

6.1 Funktionale Verifikation

6.1.1 Integration der Vorwärtsfehlerkorrektur (FEC)

Wie bereits in Abschnitt 5.8 beschrieben, wurde ein Vorwärtsfehlerkorrekturmechanismus in die OFDM-Verarbeitungskette integriert. Die mittels HLS entwickelten Codierungs- und Decodierungsmodule wurden ohne Änderungen an der bestehenden Architektur über AXI-Stream-Schnittstellen in die Verarbeitungsstrecke integriert, wodurch eine direkte Anbindung an die bestehenden OFDM-Module möglich war.

Die Verifikation erfolgte im blockorientierten Testmodus, da eine Echtzeitverarbeitung für die initiale Funktionsprüfung nicht erforderlich war. Es wurden synthetische Datenblöcke mit Hilfe von Matlab-Modell generiert, über die vollständige Sender- und Empfängerkette gesendet und am Ausgang des Decoders mit den ursprünglich eingespeisten Daten verglichen. Die vollständige Übereinstimmung der Ausgangsdaten mit den Referenzwerten bestätigte die korrekte Funktion der Codierungs- und Decodierungslogik. Damit konnte die prinzipielle Funktionsfähigkeit der integrierten FEC-Einheit nachgewiesen werden.

6.1.2 Audioverifikation im Echtzeitbetrieb

Gemäß Abschnitt 5.9 wurde die funktionale Verifikation der Audioein- und -ausgabe durchgeführt, mit dem Ziel, die korrekte Übertragung eines analogen Eingangssignals durch die vollständige OFDM-Verarbeitungskette nachzuweisen. Die Validierung wurde sowohl durch Oszilloskopmessungen als auch durch die Analyse des Amplitudengangs der verarbeiteten Ausgangssignale durchgeführt.

Abbildung 6.1 zeigt die zeitliche Abfolge der Verarbeitungsschritte innerhalb der FPGA-basierten Pipelining-Architektur. Dargestellt sind die Abschnitte zur Datenerfassung, zur eigentlichen Verarbeitung (*Proc*) sowie zur Ausgabe der Daten. Das grün hervorgehobene Segment stellt das Leerlaufzeitfenster des FPGAs dar, währenddessen dieser auf neue Eingangsdaten wartet. Dieses Zeitfenster ist abhängig vom internen Takt des FPGAs und verdeutlicht die Synchronisation zwischen Verarbeitung und Datenbereitstellung. Die dargestellte Ablaufsequenz entspricht mit den in Abschnitt 5.6.2 dokumentierten Zeitwerten, deren Gültigkeit durch die Oszilloskopmessungen verifiziert wurde.

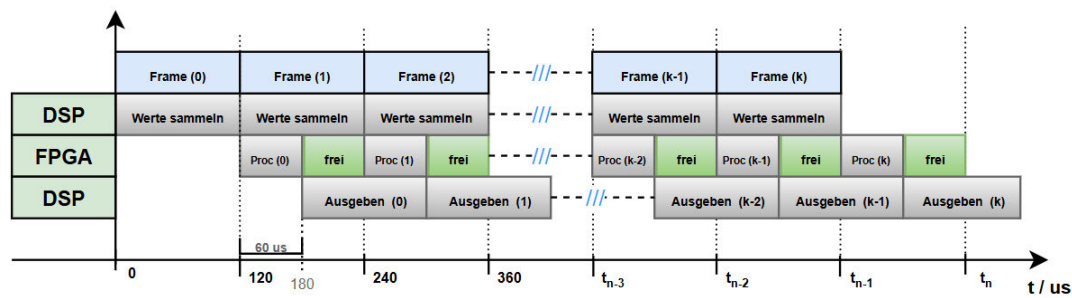


Abbildung 6.1: Zeitlicher Ablauf der Verarbeitung in der Pipelining-Struktur mit verdeutlichtem FPGA-Leerlaufzeitfenster

Für die quantitative Latenzanalyse wurde ein analoges Rechtecksignal mit einem Tastverhältnis von 1:4 (20 % High, 80 % Low) an den ADC-Eingang angelegt. Die Signalverläufe an verschiedenen Verarbeitungsstufen wurden mit einem Oszilloskop aufgezeichnet und in Abbildung 6.2 dargestellt:

- **CH1** zeigt das eingespeiste Signal am ADC-Eingang.
- **CH2** stellt das Ausgangssignal dar, das ohne Verarbeitung direkt durchgeschleift wurde.
- **CH3** zeigt das Ausgangssignal nach vollständiger Verarbeitung über die OFDM-Sende- und Empfangspfadkette.

Die Messung zeigt eine konstante, systembedingte ADC/DAC-Latenz von ca. 130 μs . Zusätzlich wurde eine Verzögerung von etwa 180 μs zwischen CH2 und CH3 gemessen, die durch die interne Verarbeitung verursacht wird. Diese setzt sich zusammen aus:

- einer Sammellatenz von ca. 120 μs bei einer Abtastrate von 100 kHz zur Erfassung eines Datenblocks (12 Abtastwerte),
- sowie einer gemessenen Verarbeitungszeit von ca. 60 μs bei einer FPGA-Taktfrequenz von 100 MHz.

Die ermittelte Gesamtverzögerung von etwa 180 μs entspricht den in Abschnitt 5.6.2 dokumentierten Messwerten.

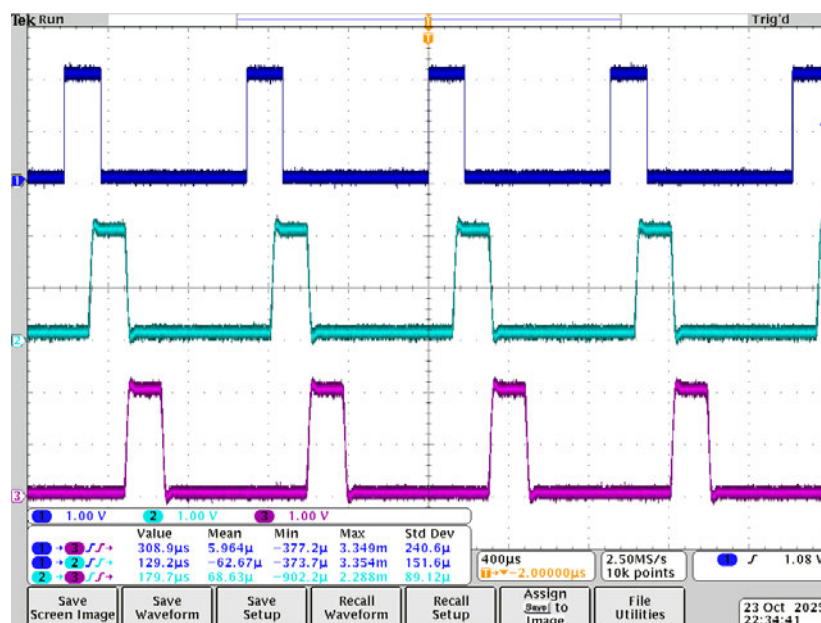


Abbildung 6.2: Oszilloskopmessung zur Latenzanalyse: Eingangssignal (CH1), durchgeschliffenes Signal (CH2) und verarbeitetes Signal (CH3)

In Abbildung 6.3 ist der gemessene Amplitudengang eines eingespeisten Sweep-Signals im Frequenzbereich von 10 Hz bis 50 kHz dargestellt. Kanal 1 (Blau) zeigt das Ausgangssignal, das ohne digitale Verarbeitung direkt durch die analoge Signalkette des DSP-Boards geschleift wurde. Kanal 2 (Rot) zeigt denselben Amplitudengang, jedoch nach vollständiger Verarbeitung durch den OFDM-Sender- und Empfängerpfad. Die nahezu identische Signalform in beiden Kanälen bestätigt die korrekte Funktion der digitalen Verarbeitungskette. Gleichzeitig ist das tiefpassartige Verhalten der analogen Ausgangsstufe des DSP-Boards deutlich erkennbar.

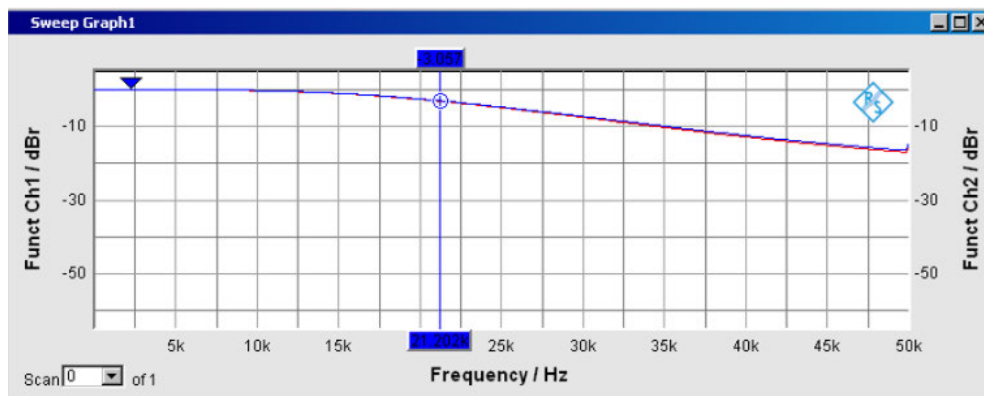


Abbildung 6.3: Amplitudengang eines Sweep-Signals (10–50kHz) am DAC-Ausgang: Kanal 1 (Blau) ohne digitale Verarbeitung, Kanal 2 (Rot) nach Durchlauf der OFDM-Verarbeitungskette.

6.2 Leistungsbewertung und Vergleich

6.2.1 Durchsatz und Latenz

In diesem Abschnitt werden die Ergebnisse hinsichtlich Durchsatz, Latenz und Ressourcenauslastung des implementierten Systems präsentiert. Die Bewertung erfolgt anhand quantitativer Messwerte und Vergleichsdaten, die sowohl für die FPGA-beschleunigte als auch für die rein DSP-basierte OFDM-Implementierung erhoben wurden.

Die maximale Datenübertragungsrate über die EMIFA-Schnittstelle ergibt sich bei einer synchronen Anbindung auf FPGA-Seite und einer Buszugriffszeit von $t_{\text{EMIFA_Zyklus}} = 60 \text{ ns}$. Zur Minimierung von Zugriffsunterbrechungen kommt dabei die EDMA zum Einsatz (vgl. Abschnitt 5.3.2).

Im Fall von Schreibzugriffen ist die Buspausenzeit während der blockweisen Übertragung vernachlässigbar. Daraus ergibt sich ein effektiver Taktzyklus von $t_{\text{eff}} \approx t_{\text{EMIFA_Zyklus}} = 60 \text{ ns}$ woraus sich gemäß Gleichung 2.20 eine maximale Schreibdatenrate von

$$D_S = \frac{16 \text{ Bit}}{60 \text{ ns}} \approx 266.7 \text{ Mbit/s}$$

ergibt.

Bei Lesezugriffen wirkt sich eine zusätzliche Buspause negativ auf die Datenrate aus. Beobachtungen zeigen eine maximale Buspause von 80 ns nach je acht übertragenen Da-

tenwörtern. Dies entspricht einer mittleren Pausenzeit von $t_P = \frac{80 \text{ ns}}{8} = 10 \text{ ns}$ pro Zugriff. Der effektive Zugriffszyklus ergibt sich somit zu

$$t_{\text{eff}} = t_{\text{EMIFA_Zyklus}} + t_P = 60 \text{ ns} + 10 \text{ ns} = 70 \text{ ns},$$

woraus sich gemäß Gleichung 2.20 eine Lese-Datenrate von

$$D_L = \frac{16 \text{ Bit}}{70 \text{ ns}} \approx 228.6 \text{ Mbit/s}$$

ergibt.

Wie in Abschnitt 5.3.2 beschrieben, tritt die Buspause regelmäßig nach acht übertragenen Wörtern auf. Der mittlere Einfluss dieser Pausen wurde in obiger Berechnung berücksichtigt.

Wie bereits in Abschnitt 5.7 beschrieben, wurde das OFDM-System zusätzlich in einer rein DSP-basierten Variante implementiert, um eine Vergleichsbasis zur hardwarebeschleunigten Lösung zu schaffen. Dabei kamen identische Systemparameter gemäß Tabelle 5.6 zum Einsatz. Die Verarbeitungszeiten wurden für beide Plattformen gemessen. Die Ergebnisse sind in Tabelle 6.1 zusammengefasst.

Plattform	t_{IFFT}	$t_{\text{Modulation}}$	t_{FFT}	$t_{\text{Demodulation}}$	t_{Gesamt}
DSP	261	21	223	16	521
FPGA	18.73		18.29		37

Tabelle 6.1: Vergleich der Verarbeitungszeiten zwischen DSP- und FPGA-basiertem OFDM-System (Angaben in μs)

Die Rechenzeiten für FFT- und IFFT-Operationen im DSP-System stellen den dominanten Anteil der Gesamtlatenz dar und verdeutlichen die begrenzte Eignung der DSP-Architektur für stark parallelisierbare Aufgaben. Die Nutzung des FPGAs als Co-Prozessor reduziert die Verarbeitungslatenz signifikant – trotz zusätzlicher EMIFA-Zugriffe. Dadurch ergibt sich eine deutliche Effizienzsteigerung in Bezug auf die Rechenzeit des Gesamtsystems.

Ergänzend zur tabellarischen Auswertung in Tabelle 6.1 zeigt Abbildung 6.4 die gemessenen Rechenzeiten der FFT-Operationen auf dem DSP und dem FPGA in Abhängigkeit von der FFT-Länge. Für jede FFT-Größe wurde die Ausführungszeit jeweils direkt auf

der Zielplattform gemessen und logarithmisch aufgetragen, um die Skalierungseffekte deutlich hervorzuheben.

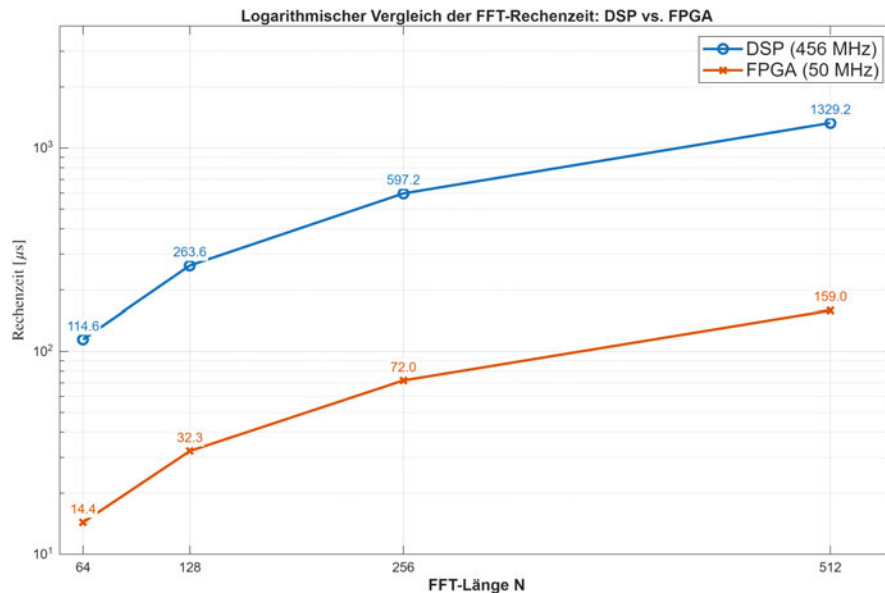


Abbildung 6.4: Vergleich der Rechenzeiten für verschiedene FFT-Längen auf DSP und FPGA (logarithmische Darstellung)

Die Abbildung 6.4 verdeutlicht eindrucksvoll die Effizienzsteigerung durch den Einsatz des FPGAs als Hardwarebeschleuniger. Die Rechenzeit wächst mit zunehmender FFT-Länge in beiden Systemen etwa proportional. Allerdings steigt sie auf dem DSP deutlich schneller an als auf dem FPGA, was die Vorteile der parallelen Verarbeitung auf FPGA-Seite unterstreicht. Dies unterstreicht die Eignung des verfolgten Co-Design-Ansatzes zur effizienten Umsetzung rechenintensiver Signalverarbeitungsaufgaben gemäß den Zielstellungen in Abschnitt 3.2.

Zur Analyse der maximal möglichen Taktfrequenz der auf dem FPGA implementierten Verarbeitungseinheiten wurde ein *Clocking Wizard* in das Design integriert [9]. Ziel war es, die vom auf dem Board verfügbaren Taktgeber abgeleitete Taktrate schrittweise zu erhöhen und deren Einfluss auf die Verarbeitungszeit zu untersuchen. Die Taktfrequenz wurde in 10 MHz-Schritten von 50 MHz bis maximal 130 MHz angepasst. Für jede Konfiguration wurde die effektive Verarbeitungszeit eines Datenblocks auf dem FPGA gemessen.

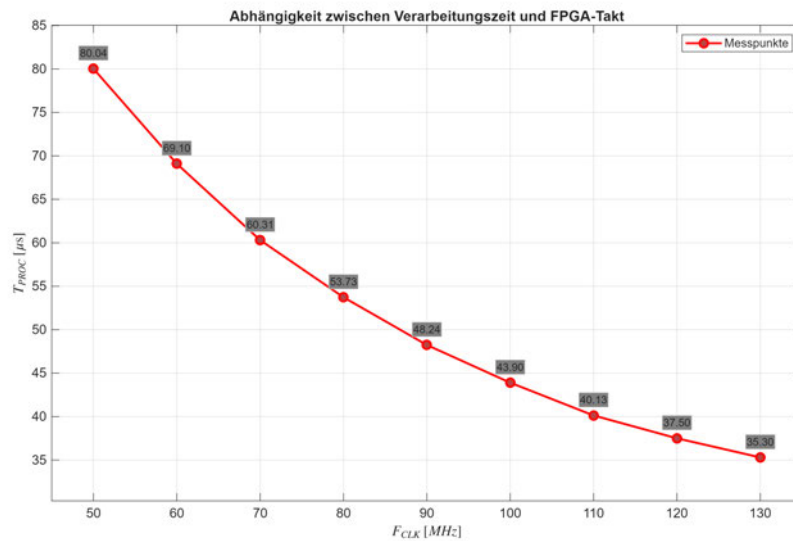


Abbildung 6.5: Abhängigkeit der Verarbeitungszeit auf dem FPGA von der Taktfrequenz

Abbildung 6.5 zeigt den gemessenen Verlauf der Verarbeitungszeit in Abhängigkeit von der Taktfrequenz. Wie zu erwarten, nimmt die Verarbeitungsdauer bei steigender Taktfrequenz deutlich ab. Der Zusammenhang ist näherungsweise invers proportional. Die höchste stabile Taktfrequenz, bei der sämtliche Timing-Anforderungen eingehalten wurden, betrug 130 MHz. Bei Frequenzen oberhalb von 130 MHz traten Setup-Verletzungen im Timing-Report auf, weshalb diese Frequenz als maximale systemstabile Taktrate definiert wurde.

6.2.2 Ressourcenauslastung auf dem FPGA

Die Analyse der Ressourcenauslastung wurde nach erfolgreicher Synthese und Implementierung durchgeführt. Untersucht wurden die belegten Komponenten im Sender- und Empfängerpfad, einschließlich der FIFO-Pufferstrukturen für Ein- und Ausgabedaten sowie der EMIFA-Schnittstelle. Während Abschnitt 5.5.3 die von Vitis HLS geschätzten Ressourcenverbräuche aufführt, enthält Tabelle 6.2 die tatsächlich gemessenen Werte nach physikalischer Implementierung auf dem Ziel-FPGA. Betrachtet wurden drei Konfigurationen mit FFT-Längen von 64, 128 und 256 Punkten.

Der Vergleich zeigt signifikante Abweichungen zwischen den durch Vitis HLS prognostizierten und den tatsächlich gemessenen Ressourcennutzungen, insbesondere bei den Lo-

Resource	Verfügbare Ressourcen	FFT 64	FFT 128	FFT 256
LUT	63400	2814	3898	4886
LUTRAM	19000	84	182	233
FF	126800	2639	4772	5325
BRAM	135	8.5	8.5	8.5
DSP	240	8	8	8
IO	285	39	39	39
MMCM	6	1	1	1

Tabelle 6.2: FPGA-Ressourcenauslastung für verschiedene FFT-Längen

gikressourcen (LUTs und Flip-Flops). Auffällig ist, dass die Anzahl belegter DSP-Blöcke für alle betrachteten FFT-Längen konstant bei acht bleibt. Dies deutet auf eine intern gepipelinte Architektur der FFT-Implementierung hin, in der die DSP-Slices mehrfach verwendet werden.

Im Gegensatz dazu steigt der Bedarf an Logikelementen mit zunehmender FFT-Länge deutlich an. Dieser Anstieg ist auf den wachsenden Bedarf an temporären Speicherelementen sowie an Steuerlogik zur Koordination der Datenströme innerhalb der FFT-Module und der angrenzenden Verarbeitungsböcke zurückzuführen. Die BRAM- und IO-Nutzung bleibt über alle Konfigurationen hinweg konstant, was auf eine effiziente und skalierungsunabhängige Nutzung dieser Ressourcenkomponenten hindeutet.

6.2.3 Syntheseergebnisse in Vivado

Die in Vitis HLS entwickelten OFDM-Sender- und -Empfängerpfade wurden in AMD Vivado synthetisiert und analysiert. Aufgrund des Umfangs der generierten Synthesebereiche kann eine vollständige Darstellung an der Stelle nicht erfolgen. Die Syntheseergebnisse von Sende- und Empfängerpfaden sind im elektronischen Anhang dokumentiert.

In der *Schematic*-Ansicht von Vivado lassen sich die aus der HLS-Entwicklung generierten IP-Module eindeutig als funktionale Einheiten identifizieren. Die modulare Strukturierung der Implementierung ermöglicht eine klare Nachvollziehbarkeit der logischen Verarbeitungsschritte, wie z. B. FFT, Modulation und Demodulation, und unterstützt damit sowohl die Systemanalyse als auch das Debugging.

Ein wesentlicher Vorteil der blockweisen Struktur besteht in der erleichterten Identifikation potenzieller Optimierungspunkte auf Ebene der Verarbeitungseinheiten. In der Synthese zeigt sich, dass bei geeigneter Strukturierung eine effiziente Abbildung der IP-Module möglich ist. Dies bildet zugleich die Grundlage für gezielte Optimierungsmaßnahmen, insbesondere im Hinblick auf die erreichbare maximale Taktfrequenz.

Die von Vitis HLS bereitgestellte automatische Optimierung versucht, die Architektur auf eine vorgegebene Ziel-Frequenz hin auszurichten. Dabei werden Scheduling Entscheidungen und Ressourcenzuweisungen so getroffen, dass kritische Pfade verkürzt und Timing-Anforderungen eingehalten werden. Diese automatische Optimierung ist jedoch nur eingeschränkt wirksam: Bei komplexeren Designs oder hohen Frequenzzielen stößt der Ansatz an seine Grenzen.

Führt dies nicht zur angestrebten Systemfrequenz, ist eine algorithmische Umstrukturierung erforderlich. Dazu zählt insbesondere der gezielte Einsatz von HLS-Pragmas (vgl. Abschnitt 5.5.1), um beispielsweise durch `#pragma HLS pipeline` oder `unroll` die Parallelisierung auf Schleifenebene zu erzwingen. Führt dies nicht zur gewünschten Performance, ist eine algorithmische Umstrukturierung erforderlich. Typische Maßnahmen umfassen dabei die Abtrennung voneinander unabhängiger Schleifeniterationen zur verbesserten Parallelverarbeitung.

6.3 Zusammenfassung der Ergebnisse

Die in diesem Kapitel präsentierten Ergebnisse belegen die erfolgreiche Umsetzung, Verifikation und Leistungsanalyse des OFDM-Systems auf einer heterogenen DSP-FPGA-Plattform unter Anwendung einer HLS-basierten Hardware-Software-Codesign-Methodik. Im Rahmen der funktionalen Verifikation konnte nachgewiesen werden, dass sämtliche Kernkomponenten – einschließlich der optional integrierten Vorwärtsfehlerkorrektur (FEC) – korrekt arbeiten. Die FEC-Module wurden erfolgreich in die OFDM Verarbeitungskette eingebunden und ihre Funktionalität mittels blockweiser Tests mit synthetisch generierten Referenzdaten bestätigt. Darüber hinaus konnte die vollständige Übertragung eines analogen Audiosignals im Echtzeitbetrieb verifiziert werden. Die dabei gemessenen Latenzen stimmten mit den zuvor simulierten Systemzeiten überein und wurden durch Oszilloskopmessungen validiert.

Die Leistungsanalyse zeigte deutliche Vorteile durch den Einsatz des FPGAs als Hardwarebeschleuniger. Gegenüber der rein DSP-basierten Variante konnten die Verarbeitungszeiten – insbesondere bei FFT/IFFT-Operationen – signifikant reduziert werden. Für die EMIFA-Schnittstelle wurde eine maximale Schreibdatenrate von ca. 266.7 Mbit/s und eine effektive Lese-Datenrate von ca. 228.6 Mbit/s bestimmt, wobei letzterer Wert durch beobachtete Buspausen beeinflusst wird (vgl. Abschnitt 6.2.1). Darüber hinaus konnte eine stabile Verarbeitung bis zu einer Taktfrequenz von 130 MHz ohne Verletzung der Timing-Anforderungen nachgewiesen werden. Die Ressourcenauslastung auf dem FPGA blieb über alle getesteten Konfigurationen hinweg deutlich unterhalb der verfügbaren Hardwarekapazitäten. Bei steigender FFT-Länge wurde ein proportionaler Anstieg des Verbrauchs von LUTs und Flip-Flops beobachtet, während die Belegung der DSP-Blöcke konstant blieb. Dies weist auf eine skalierbare und ressourcenschonende Architektur der HLS-basierten Module hin.

Die in Vivado ermittelten Syntheseergebnisse bestätigten die saubere strukturelle Umsetzung der Systemarchitektur sowie die klare Nachvollziehbarkeit der Datenflüsse. Darüber hinaus ermöglichen die verfügbaren Optimierungsansätze – sowohl automatisch als auch manuell über HLS-Pragmas und algorithmische Umstrukturierungen – eine flexible Anpassung und gezielte Performancesteigerung. Insgesamt zeigt die Auswertung, dass das implementierte System sowohl funktional als auch hinsichtlich Effizienz und Ressourcennutzung den in Kapitel 3 formulierten Anforderungen entspricht. Eine detaillierte Übersicht über den Erfüllungsgrad der Ziele und Anforderungen befindet sich im Anhang in den Tabellen A.1, A.2 und A.3.

7 Zusammenfassung und Ausblick

7.1 Systembewertung und Reflexion

Die vorliegende Arbeit untersucht den Einsatz einer HLS-basierten Hardware-Software-Codesign-Methodik zur Umsetzung eines modularen OFDM-Systems auf einer heterogenen DSP-FPGA-Plattform. Ziel war die Entwicklung und Evaluation eines vollständigen Basisband-Datenpfads für eine kontinuierliche Signalverarbeitung am Beispiel eines Audiosignals. Ausgehend von einem MATLAB-Referenzmodell wurde eine vollständige Sende- und Empfangskette konzipiert und umgesetzt. Rechenintensive Verarbeitungsschritte wie FFT/IFFT, Mapping und Modulation wurden in HLS auf dem FPGA realisiert, während der DSP Steuerung, Ablaufkontrolle und Datentransfer übernahm. Zur Kommunikation wurde die EMIFA-Schnittstelle eingesetzt, wobei sowohl wortweise Übertragungen über den DSP-Core als auch blockweise Transfers mittels EDMA evaluiert wurden. Ein hybrides Pufferkonzept aus asynchronen FIFOs und synchroner Buslogik stellte dabei eine stabile und taktische Datenübertragung sicher.

Die High-Level-Synthese wurde gezielt zur Umsetzung zentraler Module genutzt. Dabei kamen Optimierungsansätze wie Pipelining, Datenflusssteuerung und gezielte Architekturentscheidungen zum Einsatz. Die resultierenden IP-Blöcke wurden erfolgreich integriert und erfüllten die in der Anforderungsanalyse definierten funktionalen und nicht-funktionalen Kriterien. Die Systemverifikation bestätigte die korrekte Verarbeitung realer Audiodaten im Echtzeitbetrieb. Die gemessene Latenz reduzierte sich im Vergleich zur DSP-only-Variante um über 90 %, und die Verarbeitung blieb bis 130 MHz stabil. Auch die Ressourcennutzung blieb über alle getesteten Konfigurationen hinweg deutlich unter den verfügbaren FPGA-Kapazitäten. Die gewählte Architektur zeigte sich darüber hinaus flexibel für verschiedene FFT-Längen und Erweiterungen.

Insgesamt konnte gezeigt werden, dass sich eine HLS-basierte Codesign-Methodik in einem gemischtarchitektonischen Kommunikationssystem effizient, zuverlässig und nachvollziehbar umsetzen lässt. Die entwickelte Architektur erfüllt die Anforderungen an

Performanz, Skalierbarkeit und Modularität und bietet eine belastbare Grundlage für weiterführende Optimierungen und Anwendungen.

7.2 Ausblick

Das entwickelte System bildet die Grundlage für weiterführende Entwicklungen im Bereich der eingebetteten Signalverarbeitung. Durch den modularen Aufbau und die klare Schnittstellendefinition lassen sich verschiedene Erweiterungen gezielt umsetzen.

Zunächst könnte das System aufbauend auf der erfolgreichen Echtzeitübertragung analoger Eingangssignale zukünftig um eine kontinuierliche Streamingverarbeitung erweitert werden. Während die aktuelle Architektur auf blockweise Pufferung setzt, eröffnet der Einsatz dynamischer Flusssteuerung die Möglichkeit, Eingangsdaten nahezu latenzfrei zu verarbeiten und die Systemeffizienz weiter zu steigern. Auch auf der Ebene der Signalverarbeitung bestehen Potenziale zur Weiterentwicklung. Die Integration adaptiver Modulationsverfahren, der Ausbau der FEC-Funktionalität oder die Ergänzung durch Kanalabschätzung und Synchronisationsmechanismen würden die Systemkomplexität realitätsnäher abbilden und den praktischen Anwendungsbezug weiter stärken. Hinsichtlich der Hardwarearchitektur ergeben sich Optimierungspotenziale insbesondere in der Gestaltung der IP-Schnittstellen. So könnte durch eine gezielte Anpassung der FIFO-Strukturen an das AXI-Protokoll die Datenübertragung zu den HLS-Modulen beschleunigt werden. Zusätzlich wäre der Einsatz ausgewählter GPIO-Pins als Handshake-Signale zwischen DSP und FPGA denkbar, um eine explizite Synchronisation und Statusrückmeldung bei der Datenübertragung zu realisieren. Für die universitäre Lehre bietet das entwickelte System eine wertvolle Ausgangsbasis für experimentelle Laborprojekte. Durch die vorhandene Parametrierbarkeit, die klare Struktur und die realitätsnahe Umsetzung können Studierende schrittweise in die Praxis des Hardware-Software-Codesigns eingeführt werden – von der Algorithmusentwicklung über HLS bis zur Systemintegration.

Schließlich könnte die Plattform durch Erweiterung der Basisbandverarbeitung an standardnahe Modulationsverfahren angepasst werden. Denkbar wäre zum Beispiel eine einfache Umsetzung ausgewählter Elemente aus 802.11- oder 5G-Standards. Dabei müsste das bestehende OFDM-Modell entsprechend angepasst und um spezifische Parameter der jeweiligen Spezifikation erweitert werden.

Literaturverzeichnis

- [1] UniDAQ Prozessor-Board UniDAQ2.DSP-ADDA D.SignT GmbH & Co. KG (Veranst.), URL <https://www.dsignt.de/de/unidaq/unidaq2-dsp-adda.html>. – Zugriff am 16.11.2025
- [2] TMS320C674x/OMAP-L1x Processor External Memory Interface A (EMIFA) Texas Instruments (Veranst.), URL https://d1.amobbs.com/bbs_upload782111/files_38/ourdev_628181CUDHZX.pdf, 2010. – Zugriff am 23.11.2025
- [3] TMS320C674x/OMAP-L1x Processor External Memory Interface A (EMIFA) User's Guide Texas Instruments (Veranst.), URL <https://www.ti.com/lit/pdf/spruf16>, 2010. – Zugriff am 16.11.2025
- [4] TMS320C6745/C6747 DSP Technical Reference Manual Texas Instruments (Veranst.), URL <https://www.ti.com/lit/ug/spruh91d/spruh91d.pdf>, 2016. – Zugriff am 16.11.2025
- [5] Vitis High-Level Synthesis User Guide (UG1399) AMD (Veranst.), URL https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf, 2025. – Version 2025.1
- [6] ABDELRAHMAN, Ahmed ; SHAMSELDIN, Mahmoud M. ; ABDELSALAM, Amr M.: Optimized Power and Speed of Split-Radix, Radix-4 and Radix-2 FFT Structures, SpringerOpen, 2024. – URL <https://asp-eurasipjournals.springeropen.com/articles/10.1186/s13634-024-01178-4>
- [7] ALKAMIL, Asia Diao E. ; HASSAN, Obey Taj A. ; HASSAN, Ahmed Hassan M. ; ABDALLA, Walaa Faisal M.: Performance Evaluations Study of OFDM Under AWGN and Rayleigh Channels. In: *2020 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9429660>, 2021, S. 1–6

- [8] AMD: Optimizing with Dataflow in Vitis HLS AMD (Veranst.), URL https://xilinx.github.io/Vitis-Tutorials/2020-1/docs/build/html/docs/Hardware_Accelerators/Design_Tutorials/01-convolution-tutorial/dataflow.html, 2020
- [9] AMD: Clocking Wizard IP Core – AMD FPGA Intellectual Property, URL <https://docs.amd.com/r/en-US/pg065-clk-wiz/Clocking-Wizard-v6.0-LogiCORE-IP-Product-Guide>, 2023. – Clocking Wizard LogiCORE IP Product Guide (PG065), Version 2022.4
- [10] AMD: Platform Cable USB II – JTAG Programmieradapter, URL <https://www.amd.com/en/products/adaptive-socs-and-fpgas/board-accessories/hw-usb-ii-g.html>, 2024. – Zugriff am 19.11.2025
- [11] AMD: Vitis High-Level Synthesis 2024.2, URL <https://www.amd.com/de/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html>, 2024. – Zugriff am 19.11.2025
- [12] AMD: Vivado Design Suite – Version 2024.2, URL <https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2024-2.html>, 2024. – Zugriff am 19.11.2025
- [13] ARM: AMBA AXI4-Stream Protocol Specification ARM (Veranst.), URL <https://developer.arm.com/documentation/ih0051/latest/>, 2010
- [14] BELLALTA, Boris: IEEE 802.11ax: High-Efficiency WLANs, IEEE, 2016. – URL <https://doi.org/10.1109/MWC.2016.7422404>
- [15] CAO, Tianbi: Implementation and Research of Vivado HLS's Function on FPGA. In: *2022 3rd International Conference on Computer Science and Management Technology (ICCSMT)*, URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10143853>, 2022, S. 240–243
- [16] CASSEAU, Emmanuel ; GAL, Bertrand L.: High-level synthesis for the design of FPGA-based signal processing systems. In: *Proceedings of the 9th International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*, IEEE, 2009. – URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5289238>

- [17] CHEN, D. ; SONG, L. ; LIU, P. ; LUO, K. ; PENG, W.: Phase Noise Estimation and Pilot Design Suppressing Intrinsic Interference for mmWave FBMC-OQAM Systems, IEEE, 2025. – URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=11087565>
- [18] CHO, Yong S. ; KIM, Jaekwon ; YANG, Won Y. ; KANG, Chung G.: Introduction to OFDM. In: *MIMO-OFDM Wireless Communications with MATLAB*, Wiley-IEEE Press, 2010, S. 111–151. – ISBN 978-0-470-82561-7
- [19] COMISKY, Dave ; AGANVALA, Sanjive ; FUOCO, Charles: A Scalable High-Performance DMA Architecture for DSP Applications. In: *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, IEEE Xplore, 2000. – URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=878317>
- [20] CONG L. N., Pierre D. ; L., Nguyen: Time synchronization algorithm in ieee 802.11a communication system, IEEE, 2012. – URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6334146>
- [21] CORPORATION, Intel: FPGA Adaptive Software Debug and Performance Analysis. In: *Intel White Paper*, URL <https://cdrdv2-public.intel.com/650308/wp-01198-fpga-software-debug-soc.pdf>, 2023
- [22] CROCKETT, Louise H. ; ELLIOT, Ross A. ; ENDERWITZ, Martin A. ; STEWART, Robert W.: Spotlight on High-Level Synthesis. In: *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*, Strathclyde Academic Media, 2014. – URL <http://www.zynqbook.com>. – ISBN 978-0992978709
- [23] DABROWSKI, Marcin ; MODELSKI, Jozef: DVB-T2 versus DVB-T what has been changed and why?, IEEE, 2008. – URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5967610>
- [24] DIECKMANN, Karsten: Schnelle Spektralanalyse auf einem DSP-Bord mittels optimiertem FFT-Code, HAW Hamburg, 2010. – URL https://reposit.haw-hamburg.de/bitstream/20.500.12738/9876/1/Bachelorthesis_K.Dieckmann.pdf
- [25] EZEUKO, Arinze: OFDM — Orthogonal Frequency Division Multiplexing, URL <https://medium.com/@ezeuko.arinze/ofdm-orthogonal-frequency-division-multiplexing-dc027be46ef2>, 2022. – Zugriff am 10.11.2025

- [26] FERNANDEZ, Pablo F. ; CASTELLS-RUFAS, David ; CARRABINA, Jordi: Hardware-Software Codesign of Wireless Transceivers on Zynq Heterogeneous Systems, IEEE, 2018
- [27] GALANIS, M. D. ; MILIDONIS, A. ; THEODORIDIS, G. ; SOUDRIS, D. ; GOUTIS, C. E.: A Partitioning Methodology for Accelerating Applications in Hybrid Reconfigurable Platforms. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, IEEE, 2007, S. 1–6
- [28] GAO, Sheng ; LI, Jian ; ZHANG, Wen ; LIU, Lei: High-Speed Image Processing and Data Transmission Based on Vivado HLS and AXI4-Stream Interface, URL <https://thesai.org/Publications/ViewPaper?Volume=11&Issue=4&Code=IJACSA&SerialNo=46>, 2020
- [29] GAY, Matthias ; LAMPE, Alexander ; BREILING, Marco: PAPR Reduction in OFDM Using Selected Mapping with Adaptive Clipping at the Transmitter, and Sparse Reconstruction at the Receiver. In: *OFDM 2014; 18th International OFDM Workshop 2014 (InOWo'14)*, IEEE, 2014, S. 1–8. – URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6924266>
- [30] GE, Hu ; YUAN, Qiao ; ZHANG, Yuhao ; SONG, Yukun ; LI, Zhenmin: Design of a Data Transmission Control Unit in a Multi-core DSP System. In: *Proceedings of the 2023 IEEE 5th International Conference on Computer Science and Educational Informatization (CSEI)*, IEEE Xplore, 2023. – URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10395998>
- [31] GERJE, Manuel: Entwicklung eines Simulationsmodells für ein Mobilfunkübertragungssystem mit Orthogonal Frequency Division Multiplexing (OFDM), HAW Hamburg, 2010. – URL https://reposit.haw-hamburg.de/bitstream/20.500.12738/5199/1/Diplomarbeit_Manuel_Gertje.pdf
- [32] GHORPADE, S. S. ; SANKPAL, S. V.: Behavior of OFDM System Using MATLAB Simulation, 2013
- [33] INSTRUMENTS, Texas: TMS320C620x/TMS3206701 DMA and CPU: Data Access Performance, URL <https://application-notes.digchip.com/001/1-1899.pdf>, 2000
- [34] INSTRUMENTS, Texas: TMS320C64x DSP Two-Level Internal Memory Reference Guide, URL <https://www.ti.com/lit/ug/spru610/spru610.pdf>, 2006

- [35] KHLIFI, Abdelhakim ; BOUALLEGUE, Ridha: Comparison between Performances of Channel Estimation Techniques for CP-LTE and ZP-LTE Downlink Systems, URL <https://arxiv.org/pdf/1208.2060>, 2012, S. 223–233
- [36] KUMAR, Ashish ; SINGH, R. P. ; SINGH, K. N.: FFT Radix-2 and Radix-4 FPGA Acceleration Techniques Using HLS and HDL for Digital Communication Systems. In: *Proceedings of the 2018 International Conference on Recent Innovations in Electrical, Electronics and Communication Engineering (ICRIEECE)*, IEEE, 2018. – URL <https://ieeexplore.ieee.org/document/9009395>
- [37] LAWRENCE, O. ; OKONBA, Brown: Analysis of Convolutional Encoder with Viterbi Decoder for Next Generation Broadband Wireless Access Systems, 2015, S. 466–468
- [38] LI, Ye (. ; STÜBER, Gordon L.: Orthogonal Frequency Division Multiplexing for Wireless Communications, Springer, 2006. – ISBN 978-0-387-23162-6
- [39] LU, Xuming ; WEN, Weijie ; TAN, Hongzhou: A Hardware/Software Testbed for the Design of OFDM Systems. In: *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, Atlantis Press, 2013 (Advances in Intelligent Systems Research). – URL <https://doi.org/10.2991/iccsee.2013.164>. – ISBN 978-90-78677-61-1
- [40] MAJUMDER, A. ; CAFFERY J, Jr.: Power line communications, URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1343222>, 2004, S. 4–8
- [41] MANUSANI, S. K. ; KSHETRIMAYUM, R. S. ; BHATTACHARJEE, R.: Robust Time and Frequency Synchronization in OFDM based 802.11a WLAN Systems. In: *2006 Annual IEEE India Conference (INDICON)*. New Delhi, India : IEEE, 2006, S. 1–4
- [42] MATHWORKS: MATLAB R2025a – Neueste Funktionen, URL https://de.mathworks.com/products/new_products/latest_features.html, 2025. – Zugriff am 19.11.2025
- [43] MONTGOMERY, Andrew: viterbi_decoder_712: Viterbi decoder and convolutional encoder for the 7,1,2 [171 133] polynomial, URL https://github.com/Andrew-Montgomery/viterbi_decoder_712. – MIT License, hard-decision Viterbi with Hamming distance metric
- [44] MOUSAVI, H. ; AMIRI, I. S. ; MOSTAFAVI, M. A. ; CHOON, C. Y.: LTE physical layer: Performance analysis and evaluation, 2019, S. 34–44

- [45] NUSS, Benjamin: Frequenzkamm-basiertes breitbandiges MIMO-OFDM-Radar, KIT Scientific Publishing, 2022. – Karlsruher Forschungsberichte aus dem Institut für Hochfrequenztechnik und Elektronik, Band 99. – ISBN 978-3-7315-1203-5
- [46] PARK, Sang U. ; KIM, Tae P. ; LEE, Mee Z. ; CHO, Yong B.: Method of RTL Debugging When Using HLS for HW Design : Different Simulation Result of Verilog VHDL. In: *2018 International SoC Design Conference (ISODC)*, URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8649954>, 2018, S. 273–274
- [47] PATIL, P. ; BHUSARE, S. ; PAWAR, S. ; PATIL, A.: Design and Verification of AXI4-Stream to FIFO Bridge Communication Protocol. In: *Proceedings of the International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, IEEE, 2022. – URL <https://ieeexplore.ieee.org/document/10055327>
- [48] PIMENTEL, Andy D. ; ERBAS, C. ; DEPRETTERE, E. F.: Using TLM for Exploring Bus-based SoC Communication Architectures. In: *Proceedings of the IEEE International Symposium on System-on-Chip (SoC)*. Tampere, Finland : IEEE, 2002. – URL <https://ieeexplore.ieee.org/document/1173040>
- [49] QINGZHONG, Jia ; XINGDOU, Wang: Research on high-speed communication technology between DSP and FPGA. In: *2016 7th IEEE Control and System Graduate Research Colloquium (ICSGRC)*, URL <https://ieeexplore.ieee.org/document/7813302>, 2016, S. 62–66
- [50] RAHMATALLAH, Yasir ; MOHAN, Seshadri: Peak-To-Average Power Ratio Reduction in OFDM Systems: A Survey And Taxonomy, IEEE, 2013. – URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6476061>
- [51] REICHARDT, Bernd: VHDL-Simulation und -Synthese, De Gruyter Oldenbourg, 2020. – ISBN 978-3-11-067345-6
- [52] REINERT, Marc: FPGA-basierte Realisierung eines OFDM-Funkübertragungssystems, CUVILLIER VERLAG, 2009. – ISBN 978-3-86955-022-0
- [53] SKLAR, Bernard ; HARRIS, Fredric J.: Multiplexing and Multiple Access. In: *Digital Communications: Fundamentals and Applications*, Pearson, 2021. – ISBN 978-0-13-458856-8
- [54] TEXAS INSTRUMENTS: Code Composer Studio 12.8.1 – Download, URL <https://www.ti.com/tool/download/CCSTUDIO12.8.1>, 2024. – Zugriff am 19.11.2025

- [55] TEXAS INSTRUMENTS / SPECTRUM DIGITAL: XDS200 USB Debug Probe – Datenblatt, URL <https://datasheet.octopart.com/TMDSEMU200-U-Texas-Instruments-datasheet-28360601.pdf>, 2024. – Zugriff am 19.11.2025
- [56] ULEMA, Mehmet: Long-Term Evolution (LTE). In: *Fundamentals of Public Safety Networks and Critical Communications Systems: Technologies, Deployment, and Management*, IEEE, 2019. – URL <https://ieeexplore.ieee.org/document/8591932>
- [57] WARKENTIN, Lukas: Orthogonal Frequency-Division Multiplexing: Grundlagen und Peak-to-Average Power Ratio Reduktionsverfahren, HAW Hamburg, 2021. – URL https://reposit.haw-hamburg.de/bitstream/20.500.12738/14519/1/Bachelorarbeit_Lukas_Warkentin_geschw%C3%A4rzt.pdf
- [58] WERNER, Martin: Information und Codierung: Grundlagen und Anwendungen, Vieweg+Teubner Verlag, 2008. – URL <https://link.springer.com/book/10.1007/978-3-8348-9550-9>. – ISBN 978-3-8348-0232-3
- [59] WERNER, Martin: Modulation eines sinusförmigen Trägers. In: WERNER, Martin (Hrsg.): *Grundlagen digitaler Nachrichtentechnik*, Springer Vieweg, Wiesbaden, 2017, S. 241–306. – URL https://doi.org/10.1007/978-3-8348-2581-0_5
- [60] WOODS, Roger ; MCALLISTER, John ; LIGHTBODY, Gaye ; YI, Ying: FPGA-based Implementation of Signal Processing Systems, John Wiley & Sons, 2017. – Also available as eTextbook: 9781119077961. – ISBN 978-1-119-077954
- [61] XILINX, AMD: Optimization Techniques in Vitis HLS AMD Xilinx (Veranst.), URL <https://docs.amd.com/r/2022.1-English/ug1399-vitis-hls/Optimization-Techniques-in-Vitis-HLS>, 2022
- [62] XILINX, AMD: Unrolling Loops to Improve Pipelining AMD Xilinx (Veranst.), URL <https://docs.amd.com/r/2022.1-English/ug1399-vitis-hls/Unrolling-Loops-to-Improve-Pipelining>, 2022. – Vitis HLS User Guide (UG1399), Version 2022.1

A Anhang

ID	Beschreibung	Priorität	erfüllt
Z1	Erfolgreiche Inbetriebnahme des Hardware-Systems mit EMIF-Verbindung zwischen DSP TMS320C6747 und FPGA AMD Artix-7.	1	✓
Z2	Auswahl, Parametrisierung, Realisierung und Validierung eines Kommunikationssystems (Sender und Empfänger) für Audiosignale basierend auf OFDM.	1	✓
Z3	Aufbau eines MATLAB-Modells zur Systemsimulation und Verifikation.	1	✓
Z4	Bewertung und Optimierung der Systemarchitektur hinsichtlich Performance, Zeitverhalten (Setup-/Hold-Zeiten) und Ressourcenverbrauch im Sinne eines Hardware-Software-Codesigns.	2	✓
Z5	Entwurf und Implementierung einer flexiblen Parametrierung zur Adaption an unterschiedliche Kanalbedingungen.	3	✓
Z6	Implementierung und Validierung eines vollständigen Signalpfads mit Audioübertragung inkl. Kanal- und Quellenkodierung über das OFDM-System.	3	✓
Z7	Implementierung und Validierung der digitalen Hardwaremodule auf dem FPGA mittels High-Level-Synthese und C auf dem DSP.	1	✓

Tabelle A.1: Ziele des Projekts (1 = hoch, 2 = mittel, 3 = niedrig)

ID	Beschreibung	Priorität	erfüllt
F1	DSP-FPGA-System moduliert/demoduliert OFDM und überträgt im Basisband.	1	✓
F2	Stabile EMIF-Verbindung zwischen DSP (TMS320C6747) und FPGA (Artix-7) ist eingerichtet und wird zuverlässig betrieben.	1	✓
F3	MATLAB-Modell des OFDM-Systems wird entwickelt und zur Verifikation des Systemverhaltens simuliert.	1	✓
F4	Audiosignale werden über das OFDM-System korrekt übertragen (abzüglich Quantisierungseffekten).	1	✓
F5	System ist für verschiedene Kanalparameter (z.B. Bandbreite, Bitrate, Medium) flexibel konfigurierbar.	2	✓
F6	Implementierung eines digitalen Modulators/Demodulators als Teil des Übertragungsweges.	1	✓
F7	FEC-Mechanismus (z.B. Faltungscodes mit Viterbi-Decoder) zur Fehlererkennung/-korrektur ist implementiert.	2	✓
F8	Interrupt- und EDMA-basierte Datenverarbeitung auf dem DSP (z.B. für ADC/DAC-Zugriffe) wird genutzt.	3	✓
F9	Geeignete Quellenkodierung für Audiosignale ist implementiert.	3	✗

Tabelle A.2: Funktionale Anforderungen (1 = hoch, 2 = mittel, 3 = niedrig)

ID	Beschreibung	Priorität	erfüllt
NF1	Rechenlastverteilung zwischen DSP und FPGA zur Echtzeitverarbeitung: Parallelisierbare Algorithmen im FPGA, Kontrolllogik im DSP.	1	✓
NF2	Ausgewogenes Verhältnis zwischen Latenz, Durchsatz und Ressourcen (LUTs, BRAM, DSP-Slices); Design muss bei hoher Taktrate stabil bleiben.	2	✓
NF3	Fließkomma-Algorithmen werden in Festkomma überführt und hinsichtlich Genauigkeit, Ressourcen und Ausführungszeit bewertet.	3	✓
NF4	Dokumentation der HLS-Entwicklungsmethodik: Architektur, Schnittstellen, Pragmas, Simulation, Erkenntnisgewinn.	1	✓
NF5	Vergleich mehrerer FFT-Längen bzgl. Latenz, Ressourcen und Integration; Auswahl der geeignetsten Variante ohne Systemanpassung.	2	✓
NF6	Vollständige Dokumentation mittels Diagrammen, Datenflüssen, Schnittstellen- und Parametertabellen für Wartbarkeit und Review.	2	✓

Tabelle A.3: Nicht-funktionale Anforderungen (1 = hoch, 2 = mittel, 3 = niedrig)

A.1 BER vergleich für $N_{\text{FFT}} \in \{64, 256\}$

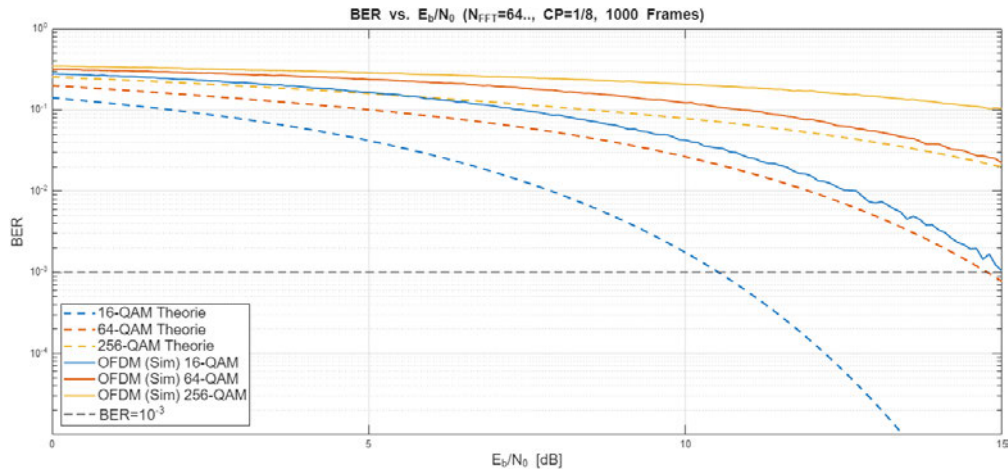


Abbildung A.1: BER gegen E_b/N_0 für 16/64/256-QAM: Theorie (AWGN) und OFDM-Simulation mit $N_{\text{FFT}} \in \{64\}$.

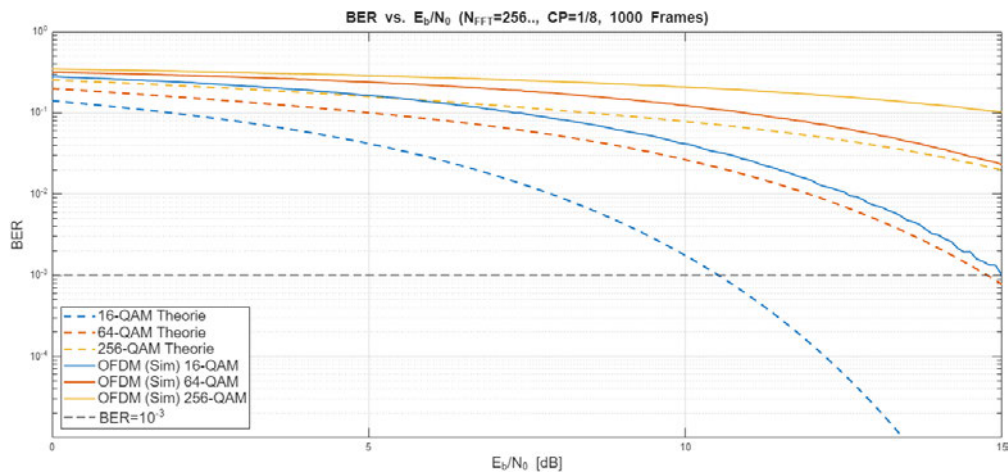


Abbildung A.2: BER gegen E_b/N_0 für 16/64/256-QAM: Theorie (AWGN) und OFDM-Simulation mit $N_{\text{FFT}} \in \{256\}$.

A.2 Vergleich der Subträgerlinien für $N_{\text{FFT}} \in \{64, 256\}$

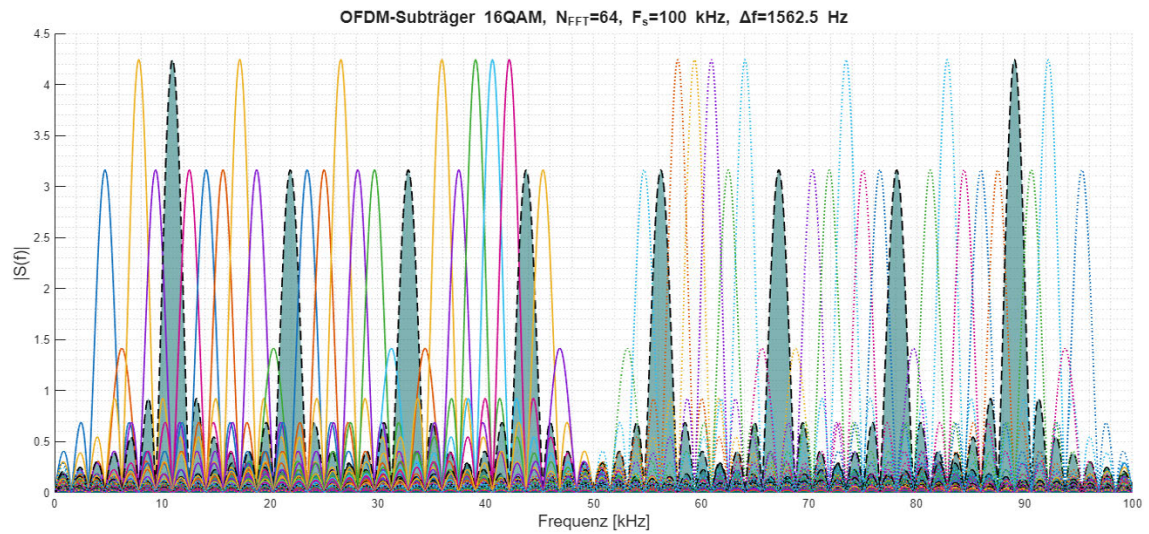


Abbildung A.3: Subträgerlinien (reelles OFDM) bei $N_{\text{FFT}} = 64$ mit Daten- und Pilotträgern.

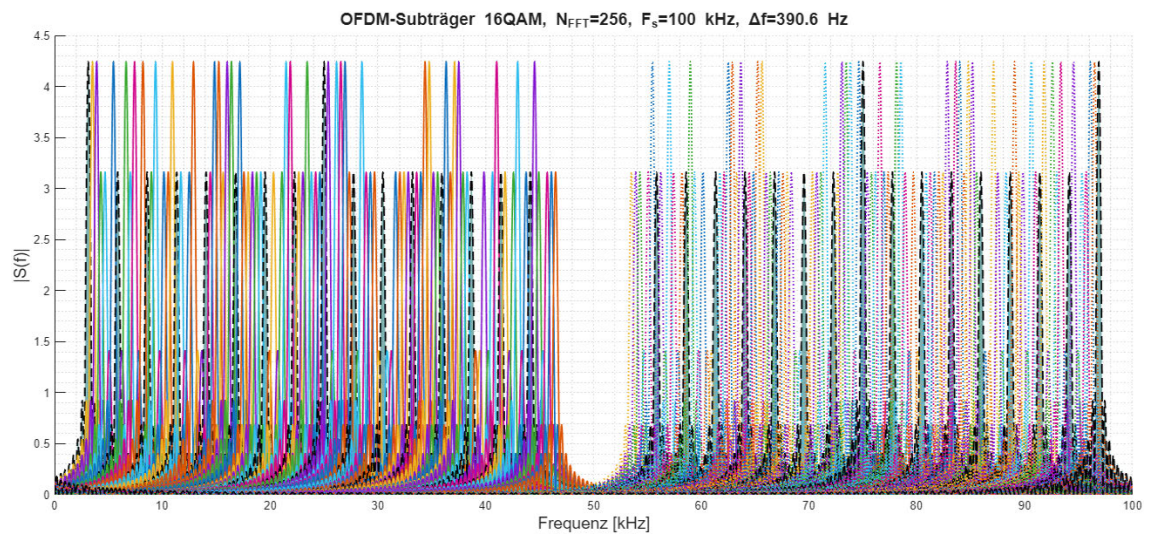


Abbildung A.4: Subträgerlinien (reelles OFDM) bei $N_{\text{FFT}} = 256$ mit Daten- und Pilotträgern.

A.3 EMIFA Messungen Schreib- Lesevorgang

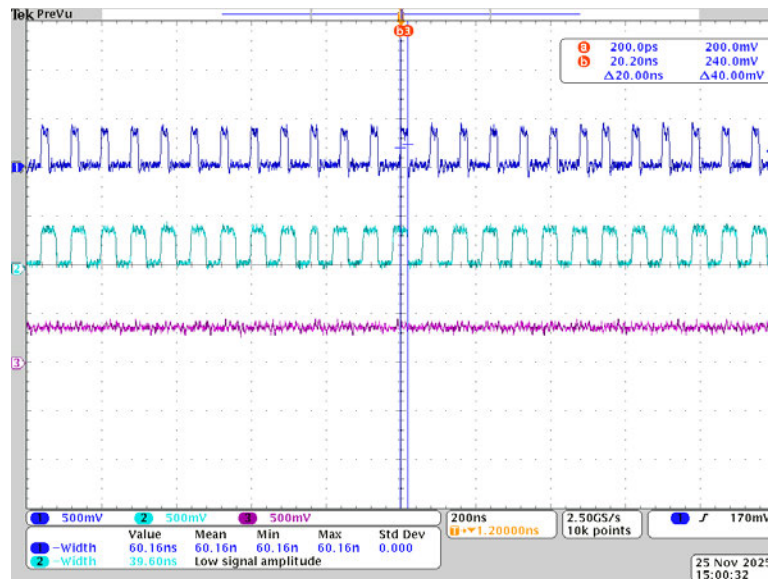


Abbildung A.5: Signalverlauf während eines Schreibvorgangs über EMIFA (DSP-Core)

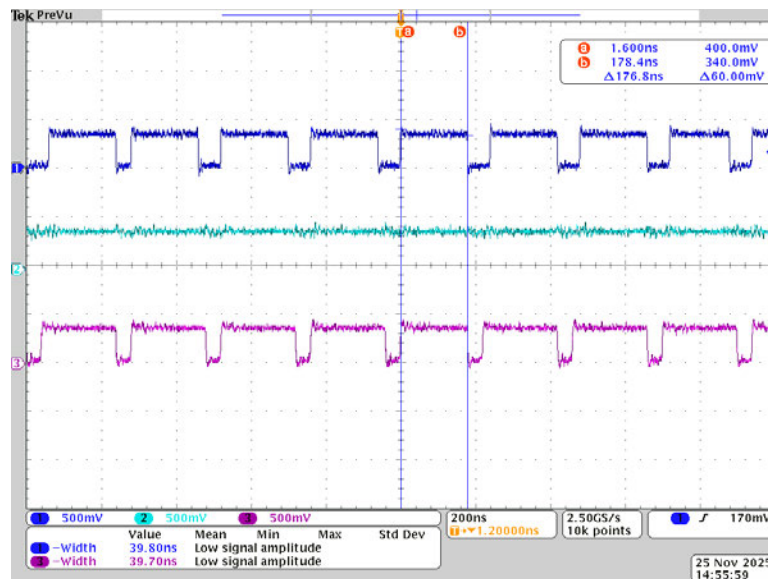


Abbildung A.6: Signalverlauf während eines Lesevorgangs über EMIFA (DSP-Core)

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original