FH Westküste
Wirtschaft und Technik

HAW HAMBURG

Masterarbeit

Knut Pröpper

Audioklassifikation mit Convolutional Neural Networks und Mel-Spektrogrammen auf einem System-on-Chip durch Hardware-Software-Co-Design und High-Level-Synthese

Fachhochschule Westküste
Fachbereich Technik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

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

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

Knut Pröpper

Audioklassifikation mit Convolutional Neural Networks und Mel-Spektrogrammen auf einem System-on-Chip durch Hardware-Software-Co-Design und High-Level-Synthese

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

Betreuender Prüfer: Prof. Dr.-Ing. Lutz Leutelt
Zweitgutachter: Prof. Dr.-Ing. Sönke Appel

Eingereicht am: 04. September 2025

Knut Pröpper

Thema der Arbeit

Audioklassifikation mit Convolutional Neural Networks und Mel-Spektrogrammen auf einem System-on-Chip durch Hardware-Software-Co-Design und High-Level-Synthese

Stichworte

System-on-Chip, Hardware-Software-Co-Design, High-Level-Synthese

Kurzzusammenfassung

In dieser Arbeit wird der Einsatz von High-Level-Synthese (HLS) im Hardware-Software-Co-Design untersucht. Zu diesem Zweck wurde eine bestehende, Python-basierte Pipeline zur Vogelstimmenklassifizierung auf ein SoC portiert. Die Implementierung erfolgt in Software, VHDL, HLS und PYNQ. HLS verkürzt die Entwicklungszeit, während HDLs bei Low-Level-Schnittstellen weiterhin überlegen sind. Die Kombination beider Ansätze ermöglicht eine effiziente Echtzeit-Klassifizierung.

Knut Pröpper

Title of Thesis

Audio classification using convolutional neural networks and Mel-spectrograms on a system-on-chip via hardware-software-co-design and high-level-synthesis

Keywords

system-on-chip, hardware-software-co-design, high-level-synthesis

Abstract

This thesis investigates the use of high-level synthesis (HLS) in hardware-software co-design. For this purpose, an existing Python-based pipeline for bird song classification was ported to an SoC. The implementation was carried out in software, VHDL, HLS and PYNQ. HLS shortens development time, while HDLs remain superior for low-level interfaces. The combination of both approaches enables efficient real-time classification.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Die stetig steigende Verfügbarkeit von Rechenressourcen und flexiblen System-on-Chip-(SoC)-Plattformen eröffnet neue Möglichkeiten für eingebettete Anwendungen in den Bereichen Signalverarbeitung und maschinelles Lernen. Eine zentrale Herausforderung besteht darin, Softwarealgorithmen effizient in Hardware umzusetzen, um die Leistungsfähigkeit und Energieeffizienz zu maximieren.

In dieser Arbeit wird der Einsatz von High-Level-Synthese (HLS) im Rahmen eines Hardware-Software-Co-Designs untersucht. Dies erfolgt am Beispiel eines Systems zur Audioklassifikation von Vogelstimmen. Aufbauend auf einem bestehenden Studierendenprojekt [16] wurde eine Python-basierte Klassifikationspipeline, die auf Convolutional Neural Networks (CNNs) und Mel-Spektrogrammen basiert, auf einen SoC übertragen.

Der Fokus liegt darauf, zu bewerten, für welche Verarbeitungsschritte HLS geeignet ist und wo die Grenzen liegen. Zu diesem Zweck werden sowohl hardware- als auch softwarebasierte Implementierungen zentraler Module entwickelt, verglichen und hinsichtlich Latenz, Ressourceneffizienz und Entwicklungsaufwand analysiert. Die gewonnenen Erkenntnisse sollen nicht nur zur Optimierung des vorliegenden Systems beitragen, sondern auch allgemeine Empfehlungen für den Einsatz von HLS in eingebetteten KI-Anwendungen ableiten.

2 Grundlagen

In diesem Kapitel werden die Grundlagen dargelegt, die für das Verständnis der vorliegenden Arbeit von essentieller Bedeutung sind. Dabei werden die folgenden Themen behandelt: Hardware-Software-Co-Design, High-Level-Synthesis, Mel-Spektrogramme, Convolutional Neural Networks, Zynq Ultrascale+ und PYNQ.

2.1 Hardware-Software-Co-Design

Hardware-Software-Co-Design ist ein koordiniertes Vorgehen, bei dem Hardware- und Software-Komponenten eines Systems gleichzeitig entworfen werden, um Leistungsfähigkeit, Energieverbrauch, Entwicklungszeit und Kosten bestmöglich auszubalancieren. Anstatt Hardware und Software getrennt zu entwickeln, werden beide Seiten iterativ aufeinander abgestimmt [39]. In [2] wird ein umfassender Überblick über Prozesse und Werkzeuge geboten.

2.1.1 Design-Flow im HW-SW-Co-Design

Ein typischer Entwicklungsprozess beginnt mit der Spezifikation, in der die Systemanforderungen definiert werden. Es folgt die Partitionierung, bei der entschieden wird, welche Funktionen in Hardware (z. B. FPGA, ASIC) und welche in Software (z. B. Embedded-Prozessor) umgesetzt werden. Danach erfolgt die parallele Implementierung: Hardware wird mit Hardwarebeschreibungssprachen oder High-Level-Synthese-Tools entwickelt, Software mit Sprachen wie C oder C++. Anschließend werden beide Komponenten integriert und gemeinsam verifiziert, häufig mithilfe von Co-Simulation [2].

2.1.2 Abstraktionsebenen und Modellierung

Um die Komplexität zu beherrschen, wird auf verschiedenen Abstraktionsebenen modelliert. Untimed Functional Modelle beschreiben Funktionen ohne Zeitinformation, Timed Functional Modelle fügen grobe Zeitabschätzungen hinzu, cycle-accurate Modelle bilden das Taktsignal exakt ab, und RTL-Beschreibungen definieren konkrete Hardwarestrukturen. SystemC wird häufig als Sprache für System-Level-Design eingesetzt, da sich Hardware- und Software-Komponenten gemeinsam simulieren und Schnittstellen früh validieren lassen [2].

2.1.3 Partitionierung und Optimierung

Die Entscheidung, welche Komponenten in Hardware oder Software realisiert werden, ist eine Kernaufgabe des Co-Designs. Kriterien sind Leistung, Energieverbrauch, Kosten und Entwicklungszeit. Funktionen mit hohem Rechenaufwand oder strikten Echtzeitanforderungen werden oft in Hardware implementiert [2].

2.1.4 Kommunikation und Schnittstellen

Effiziente Schnittstellen zwischen Hardware und Software sind entscheidend für das Gesamtsystem. Typische Beispiele sind Busprotokolle wie AXI oder AMBA, Peripherie-Schnittstellen wie SPI oder I2C sowie gemeinsame Speicherbereiche und Interrupt-Mechanismen [22].

2.1.5 Verifikation und Co-Simulation

Die Validierung erfolgt oft durch Co-Simulation, bei der Hardwaremodelle in SystemC oder VHDL und Softwaremodelle in C oder C++ gemeinsam ausgeführt werden. In [28] wird beschrieben, wie dieser Prozess automatisiert und beschleunigt werden kann.

2.2 High-Level-Synthese

Als High-Level Synthesis (HLS) wird der Prozess bezeichnet, bei dem Hardware aus einer abstrakten, algorithmischen Beschreibung, typischerweise in Sprachen wie C oder C++, automatisiert auf Register-Transfer-Level (RTL) umgesetzt wird. Das Ziel besteht darin, den Hardwareentwurf von der rein strukturellen Beschreibungsebene (VHDL/Verilog) auf eine funktionale, architektonisch abstrahierte Ebene zu heben. Dadurch werden Entwicklungszeiten verkürzt, Designänderungen erleichtert und eine explorative Optimierung unterschiedlicher Implementierungsvarianten ermöglicht [19].

2.2.1 HLS Workflow

Der typische HLS-Workflow umfasst mehrere Schritte. Zunächst wird der Algorithmus in einer Hochsprache wie C oder C++ beschrieben. Darauf folgt die *C-Simulation*, mit der die funktionale Korrektheit des Codes überprüft wird. Im nächsten Schritt transformiert der HLS-Compiler den Quellcode in eine RTL-Beschreibung, die den Entwurf in Hardwarestrukturen wie Register, Multiplexer und Zustandsautomaten überführt.

Anschließend wird eine *C/RTL-Co-Simulation* durchgeführt, um sicherzustellen, dass die RTL-Implementierung funktional identisch zur ursprünglichen Algorithmusbeschreibung ist. Auf dieser Basis können gezielte Optimierungen vorgenommen werden, etwa durch Pipelining oder Parallelisierung. Schließlich werden die RTL-Designs im FPGA-Designflow (z. B. mit Vivado) synthetisiert, implementiert und auf die Zielhardware übertragen [14].

2.2.2 Optimierungen

Für eine effiziente Umsetzung der Algorithmen in Hardware ist es notwendig, den C/C++-Code gezielt für HLS zu optimieren. Hierbei kommen sogenannte *Pragmas* zum Einsatz, mit denen der Entwickler die Abbildung auf Hardware beeinflussen kann. Im Folgenden werden drei zentrale Optimierungsstrategien beschrieben. Weitere sind z. B. in [14] zu finden.

Pipeline

Durch Pipelining können aufeinanderfolgende Operationen überlappend ausgeführt werden. Dies reduziert die Latenz pro Operation und steigert den Durchsatz erheblich. Besonders bei Berechnungen in Schleifen oder bei Datenströmen ist Pipelining essenziell, um eine hohe Taktfrequenz und parallele Ausführung zu erreichen. In Abbildung 2.1 ist dies dargestellt.

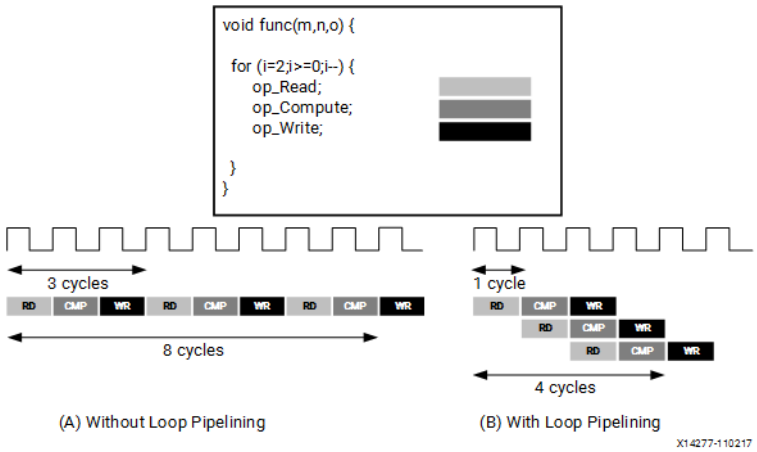

Abbildung 2.1: Optimierung durch Pipelining [14]

Loop Unroll

Bei der Unrolling-Optimierung wird der Schleifenrumpf mehrfach repliziert, sodass mehrere Iterationen einer Schleife parallel abgearbeitet werden können. Dies kann die Ausführungszeit deutlich reduzieren, erfordert jedoch zusätzliche Hardware-Ressourcen. Der Grad des Unrollings kann dabei flexibel gewählt werden, um einen Kompromiss zwischen Flächenbedarf und Geschwindigkeit zu finden. In Abbildung 2.2 ist dies dargestellt.

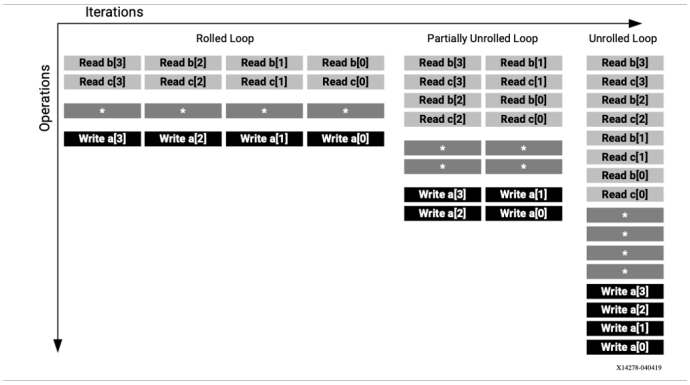

Abbildung 2.2: Optimierung durch Loop Unroll [14]

Dataflow

Mit der Dataflow-Optimierung lassen sich voneinander unabhängige Funktionsblöcke oder Schleifen parallel ausführen. Zwischen den Blöcken werden FIFO-Speicher eingefügt, wodurch eine kontinuierliche Datenverarbeitung im Streaming-Stil ermöglicht wird. Besonders für Anwendungen mit Pipelines von Verarbeitungsschritten, wie in Signal- oder Bildverarbeitungsalgorithmen, ist diese Optimierung von großer Bedeutung. In Abbildung 2.3 ist dies dargestellt.

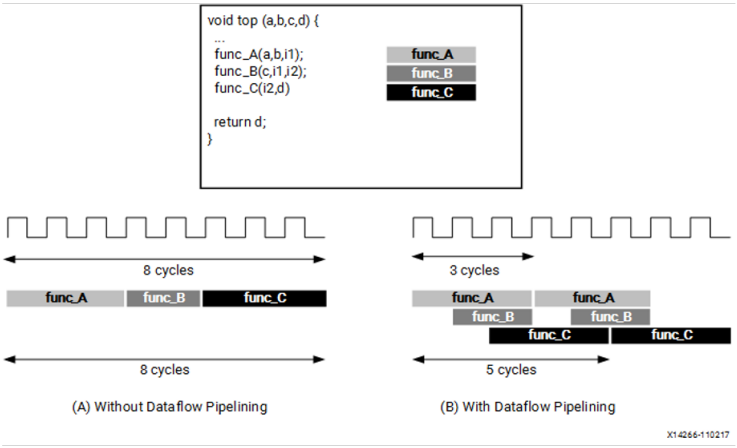

Abbildung 2.3: Optimierung durch Dataflow [14]

2.2.3 Vitis-HLS

Vitis HLS ist die von AMD/Xilinx bereitgestellte Entwicklungsumgebung für High-Level-Synthese und seit der Einführung der Vitis Unified IDE in diese integriert. Innerhalb der Vitis-Umgebung können sowohl reine Softwarekomponenten als auch HLS-basierte Hardwarebeschreibungen erstellt, simuliert und optimiert werden. Die Integration erleichtert die Kombination von Software und Hardware in heterogenen SoC-Architekturen wie dem Zynq UltraScale+ MPSoC und unterstützt damit nahtlos den hardwarebeschleunigten Entwurf [14].

2.3 Mel-Spektrogramm

Ein Mel-Spektrogramm ist eine visuelle Darstellung des Frequenzinhalts eines Audiosignals über die Zeit. Die Frequenzen werden dabei auf einer sogenannten Mel-Skala dargestellt. Die Mel-Skala basiert auf der menschlichen Wahrnehmung von Tonhöhen und ist logarithmisch, sodass gleiche Abstände auf der Skala als gleich große Tonhöhenänderungen empfunden werden. Der Aufbau eines Mel-Spektrogramms ähnelt dem Aufbau eines gewöhnlichen Spektrogramms. Die x-Achse stellt die Zeit dar, während die y-Achse die Frequenzbänder auf der Mel-Skala zeigt. Die Farbgebung oder Helligkeit im Bild geben die Energie (Lautstärke) in den jeweiligen Frequenzbändern zu einem bestimmten Zeitpunkt an. Die Abbildung 2.4 zeigt ein entsprechendes Mel-Spektrogramm [36].

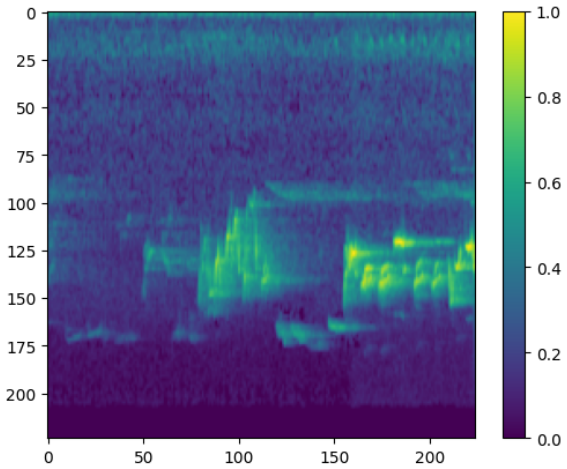

Abbildung 2.4: Mel-Spektrogramm einer Singdrossel

Die Anwendungsbereiche von Mel-Spektrogrammen umfassen die Spracherkennung (beispielsweise in digitalen Assistenten), die Sprachsynthese (beispielsweise Text-to-Speech-Systeme), die Musikerkennung oder Genreklassifikation, die Klanganalyse in der Bioakustik oder Musikforschung, das maschinelle Lernen, bei dem Modelle wie CNNs auf Mel-Spektrogramme trainiert werden, ähnlich wie auf Bilder [29, 21, 20].

Wie in Abbildung 2.5 dargestellt, wird das Mel-Spektrum in mehreren Schritten erzeugt [8]. Zunächst wird der grobe Ablauf beschrieben, anschließend werden die Funktionen der einzelnen Module in den folgenden Abschnitten erläutert.

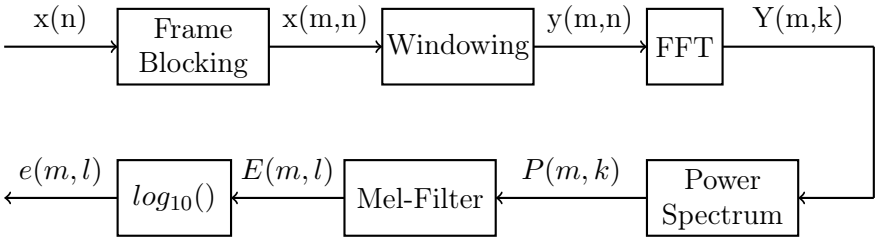

Abbildung 2.5: Blockdiagramm der Mel-Spektrogramm generierung

Das Eingangssignal $x(n)$ wird zunächst in Segmente fester Länge N unterteilt, die als *Frames* bezeichnet werden. Das so entstehende Signal $x(m, n)$ wird dabei über den Frameindex m organisiert. Anschließend wird auf jedes Segment eine Fensterfunktion der Länge N angewendet, wodurch das gefensterte Signal $y(m, n)$ entsteht [25].

Daraufhin wird mittels einer FFT das Spektrum $Y(m, k)$ berechnet und durch Quadrierung des Betrags in das Leistungsspektrum $P(m, k)$ überführt. Dieses wird mit einer Mel-Filterbank gefaltet, sodass das gefilterte Signal $E(m, l)$ entsteht, wobei l den Index der Mel-Filter bezeichnet. Abschließend wird der Amplitudenverlauf logarithmiert, was zu $e(m, l)$ führt.

Jedes $e(m, l)$ stellt einen einzelnen *Frame* des Mel-Spektrums dar. Durch Aneinanderreihen mehrerer solcher Frames ergibt sich schließlich ein vollständiges Mel-Spektrogramm.

2.3.1 Frame Blocking

Das Eingangssignal $x(n)$ wird in gleich lange Segmente (*Frames*) mit einer Länge von N unterteilt. Die einzelnen *Frames* beginnen jeweils mit einer Schrittweite (*Hop Size*) S, wobei $S < N$ gilt. Dadurch überlappen sich die *Frames* um $(N - S)$ Abtastwerte [25]. Dieses Verfahren ist in Abbildung 2.6 dargestellt.

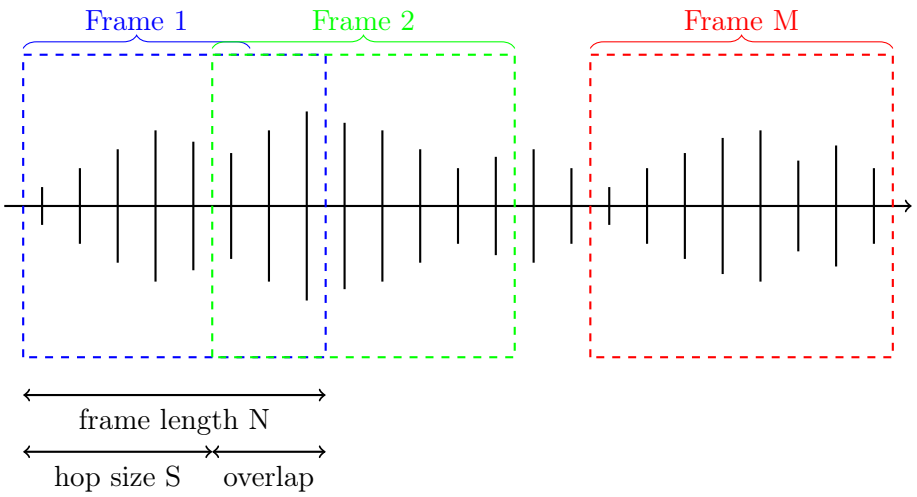

Abbildung 2.6: Diagram of frame blocking

Der erste *Frame* besteht aus den ersten N Abtastwerten, der zweite *Frame* beginnt um S Abtastwerte verschoben und so weiter. Formal gilt für den m-ten Frame:

$$x(m, n) = x(m \cdot S + n) \tag{2.1}$$

2.3.2 Windowing

Nach dem *Framing* werden die Eingangsdaten mit einem Fenster multipliziert, um Spektralleckagen bei der nachfolgenden FFT zu verringern. Durch die Fensterung werden zudem sprunghafte Übergänge an den Rahmenrändern geglättet, wodurch unerwünschte Nebeneffekte im Frequenzspektrum verringert werden [25].

$$y(n) = x(n) \cdot w(n) \tag{2.2}$$

Für Audioklassifikationsaufgaben werden häufig Hamming- oder Hanning-Fenster verwendet, da sie einen guten Kompromiss zwischen der Breite der Hauptkeule und der Unterdrückung der Nebenkeulen bieten [8, 18]. In dieser Arbeit kommt das Hanning-Fenster zum Einsatz, das sich durch die folgende Gleichung [33] definiert:

$$w(n) = 0,5 - 0,5 \cdot cos\left(\frac{2\pi n}{N-1}\right), n = 0, \ldots, N-1 \tag{2.3}$$

Die resultierende Fensterfunktion ist in Abbildung 2.7 dargestellt.

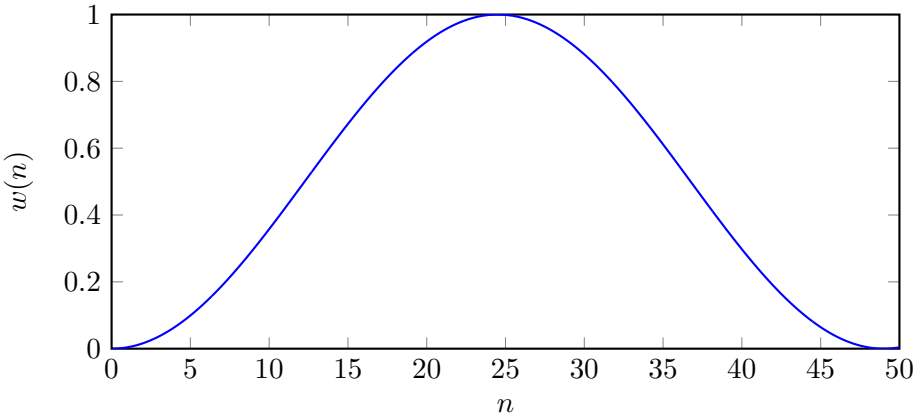

Abbildung 2.7: Hanning-Fenster

2.3.3 FFT

Anschließend wird auf dem gefensterten Signal die FFT berechnet, um es vom Zeit- in den Frequenzbereich zu überführen. Die DFT, auf der die FFT basiert, ist definiert als [33]:

$$Y(m,k) = \sum_{n=0}^{N-1} y(m,n) e^{-j\frac{2\pi}{N}kn} \tag{2.4}$$

Dabei beschreibt $y(m,n)$ das gefensterte Eingangssignal und $Y(m,k)$ die komplexen Spektralkoeffizienten. Die FFT ist ein effizienter Algorithmus zur Berechnung dieser Transformation und reduziert den Rechenaufwand erheblich [33].

2.3.4 Leistungsspektrum

Im Anschluss an die FFT wird aus den komplexen Spektralwerten das Leistungsspektrum berechnet. Da das Eingangssignal reellwertig ist, weist das berechnete Frequenzspektrum eine konjugiert-symmetrische Struktur auf. Das bedeutet, dass für jede Frequenzkomponente $Y(m,k)$ eine komplex konjugierte Komponente $Y(m,N-k)$ existiert. Folglich sind keine zusätzlichen Informationen in der zweiten Hälfte des Spektrums enthalten, sodass für weitere Berechnungen nur die erste Hälfte (bis zur Nyquist-Frequenz) benötigt wird [23].

Das Leistungsspektrum $P(m,k)$ ergibt sich aus dem Quadrat der FFT-Koeffizienten:

$$P(m,k) = |Y(m,k)|^2 = Y_r^2(m,k) + Y_i^2(m,k) \tag{2.5}$$

wobei $Y_r(m,k)$ und $Y_i(m,k)$ die Real- bzw. Imaginärteile der FFT darstellen.

2.3.5 Mel-Filter

Anschließend werden die Frequenzen in den Mel-Frequenzbereich transformiert. Dieser orientiert sich an der menschlichen Wahrnehmung von Tonhöhen und lässt sich durch die folgende Gleichung [49] beschreiben:

$$f_{mel} = 2595 \cdot log_{10}\left(1 + \frac{f_{Hz}}{700}\right) \tag{2.6}$$

Zur Berechnung der spektralen Energien in Mel-Bändern kommen Dreiecksfilter zum Einsatz. Die Gewichtungsfunktion $H_m(n)$ ist wie folgt definiert [6]:

$$H_m(n) = \begin{cases} 0 & k < f(m-1) \\ \frac{k-f(m-1)}{f(m)-f(m-1)} & f(m-1) < k < f(m) \\ \frac{f(m+1)-k}{f(m+1)-f(m)} & f(m) < k < f(m+1) \\ 0 & k > f(m+1) \end{cases} \tag{2.7}$$

Das Leistungsspektrum wird mit M solcher Dreiecksfilter gewichtet. Die Ausgangsleistung des l-ten Filters ergibt sich zu:

$$E(m,l) = \sum_{k=0}^{N-1} P(m,k) \cdot H_m(k) \tag{2.8}$$

Die resultierenden Filterbänder über dem Frequenzspektrum sind in Abbildung 2.8 dargestellt.

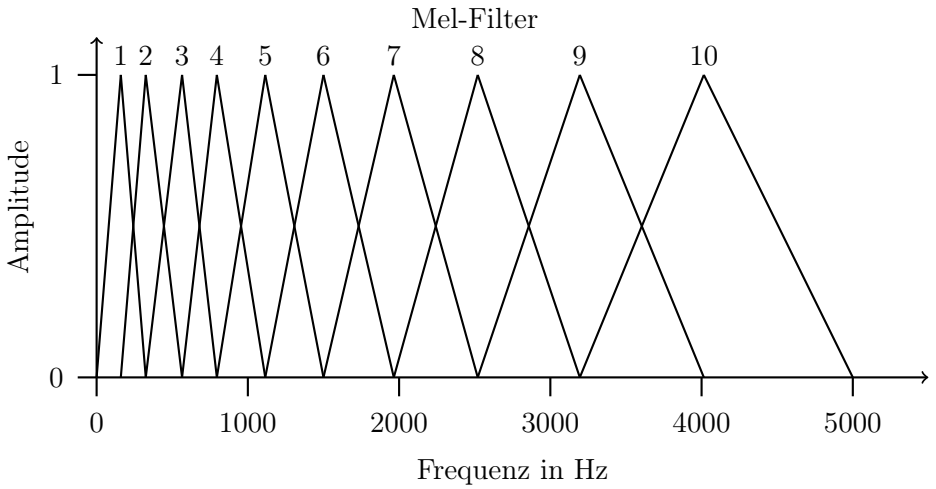

Abbildung 2.8: Mel-Filter

Da die Auflösung des Spektrums in höheren Frequenzbereichen dichter ist als in niedrigeren, müssen die Filter normalisiert werden. Dadurch wird verhindert, dass die Filter in hochfrequenten Bereichen aufgrund der größeren Anzahl von Spektralkomponenten unverhältnismäßig große Beiträge liefern [37].

2.3.6 Logarithmische Skalierung

Um den großen Dynamikbereich der berechneten Filterenergien zu verringern und die Darstellung dem menschlichen Lautstärkeempfinden anzupassen, werden die Ergebnisse logarithmisch skaliert. Dies geschieht nach folgender Gleichung [23]:

$$e(m,l) = log_{10}(E(m,l)) \tag{2.9}$$

Dadurch werden Unterschiede zwischen starken und schwachen Frequenzanteilen komprimiert, was zu einer robusteren Merkmalsdarstellung führt.

2.4 Convolutional Neural Network

Convolutional Neural Networks (CNNs) sind eine spezielle Form künstlicher neuronaler Netze, die sich besonders für die Verarbeitung von Bild- und Sensordaten eignen. Im Gegensatz zu klassischen Multilayer-Perceptrons (MLPs), die flache Eingangsvektoren erwarten, arbeiten CNNs direkt mit mehrdimensionalen Daten (z. B. Bildmatrizen) [27]. Dadurch bleiben räumliche Strukturen erhalten, und es können Merkmale unabhängig von deren Position im Eingabebild erkannt werden.

Ein Convolutional Neural Network (CNN) ist, wie in Abbildung 2.9 dargestellt, hierarchisch aufgebaut. In den ersten Convolutional Layers werden grundlegende Merkmale wie Kanten oder Texturen erkannt. In den mittleren Schichten werden diese Merkmale durch Pooling Layers verdichtet und abstrahiert, sodass komplexere Strukturen wie Formen oder Objekte entstehen. Die abschließenden Fully Connected Layers übernehmen schließlich die Klassifikation auf Basis der extrahierten Merkmale. Im Folgenden werden die einzelnen Schichttypen kurz erläutert.

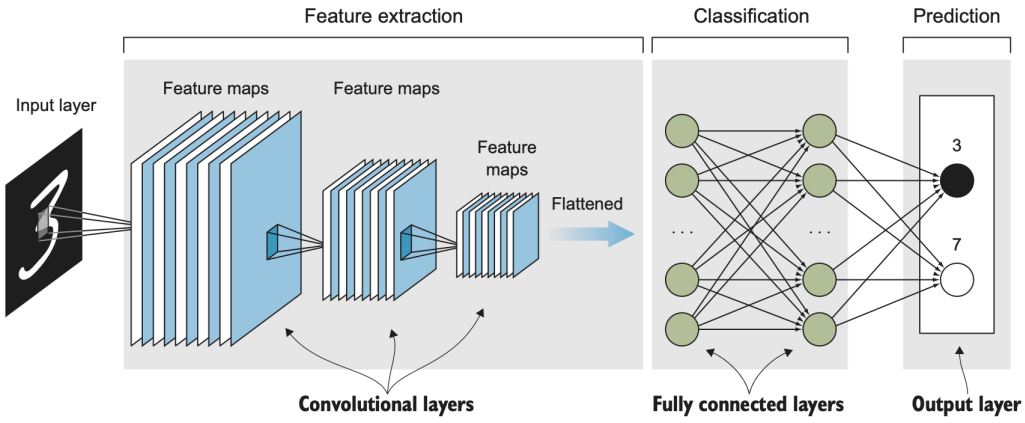

Abbildung 2.9: Struktureller Aufbau eines CNN [26]

2.4.1 Convolutional Layer

Convolutional Layer bilden das Kernstück eines CNN. Sie bestehen aus trainierbaren Filtern (z.B. 3x3 oder 5x5), die über die Eingabedaten gleiten (Faltung) und lokale Merkmale extrahieren. Jeder Filter erzeugt eine sogenannte Feature Map, welche die erkannten

Muster anzeigt. Parameter wie Stride (Schrittweite) und Padding (Auffüllen der Ränder mit Nullen) steuern die Größe der resultierenden Feature Maps [26].

2.4.2 Pooling Layer

Pooling Layers reduzieren die räumliche Auflösung der Feature Maps, um Rechenaufwand und Speicherbedarf zu verringern und gleichzeitig Translationstoleranz zu erhöhen. Häufig werden Max-Pooling oder Average-Pooling mit kleinen Fenstern (z.B. 2x2) eingesetzt. So bleiben die wichtigsten Merkmale erhalten, während unwichtige Details verworfen werden [26].

2.4.3 Fully Connected Layer

Am Ende der Faltungs- und Pooling-Operationen wird das mehrdimensionale Merkmalsfeld zu einem eindimensionalen Vektor "geflattetünd in Fully Connected Layers (klassisches MLP) überführt. Diese Schichten kombinieren die erlernten Merkmale und führen typischerweise über eine Softmax-Aktivierung eine Klassifikation aus [26].

2.4.4 Training und Optimierung

CNNs werden durch überwachte Lernverfahren trainiert. Dabei passen Backpropagation und Gradientenabstiegsverfahren die Filtergewichte so an, dass der Fehler zwischen vorhergesagtem und tatsächlichem Label minimiert wird. Um Overfitting zu vermeiden, werden Datensätze üblicherweise in Training, Validierung und Test aufgeteilt (z.B. 70 / 15 / 15). Alternativ kann Transfer Learning eingesetzt werden, bei dem ein vortrainiertes Netz an neue Aufgaben angepasst wird, um Trainingszeit und Datenbedarf zu reduzieren [26].

2.5 Zynq Ultrascale+

Der Zynq UltraScale+ MPSoC kombiniert ein Processing System (PS) und eine programmierbare Logik (PL) auf einem Chip. Das PS ist ein fest integrierter, optimierter Teil mit mehreren Verarbeitungseinheiten sowie Plattform- und Sicherheitssystemen. Die PL

basiert auf FPGA-Fabric, ist frei konfigurierbar und kann während des Betriebs neu programmiert werden, um benutzerdefinierte Logikfunktionen oder Hardwarebeschleuniger zu realisieren.

Es gibt drei Gerätefamilien – CG, EG und EV – die sich in der Ausstattung des PS und PL unterscheiden [9]:

- CG-Familie: Dual-Core-APU (Cortex-A53) + Dual-Core-RPU (Cortex-R5)
- EG-/EV-Familie: Quad-Core-APU + Dual-Core-RPU + Arm Mali GPU für Grafikbeschleunigung

Alle Familien teilen sich die gleiche Grundarchitektur, unterscheiden sich jedoch in der Leistungsfähigkeit und den verfügbaren Ressourcen. In Abbildung 2.10 ist das Blockschaltbild der EG-Variante dargestellt. Im Folgenden werden die einzelnen Komponenten des Systems im Detail beschrieben [32].

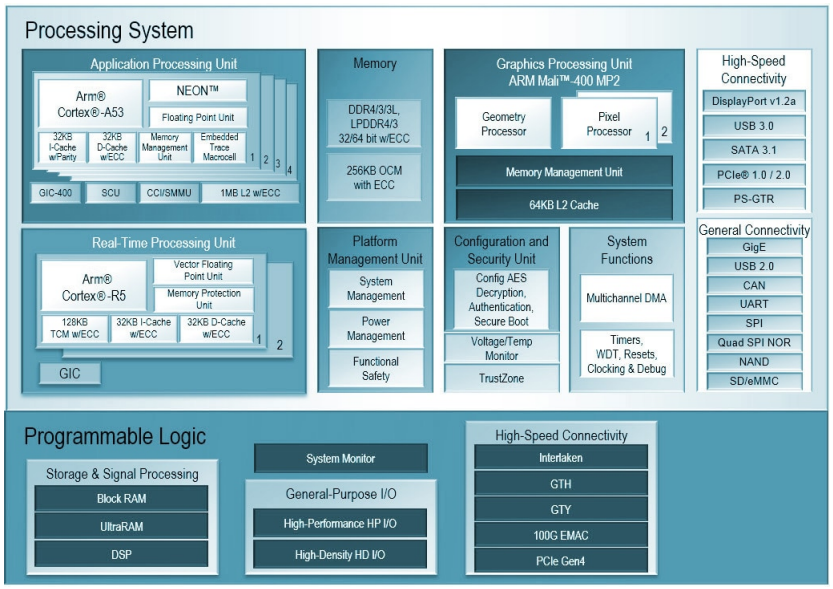

Abbildung 2.10: ZYNQ Ultrascale+ EG Blockdiagramm [9]

2.5.1 Processing System

Das Processing System (PS) des Zynq MPSoC vereint APU, RPU, GPU, externe Speichercontroller und zahlreiche Peripherieeinheiten. Alle Prozessoren sind über das Cache Coherent Interconnect (CCI) verbunden, das Datenkohärenz sicherstellt und eine dynamische Lastverteilung zwischen den Kernen ermöglicht. Die Kommunikation nach außen erfolgt über MIO für Standardperipherie und PS-GTR-Transceiver für Hochgeschwindigkeitsverbindungen. Zusätzlich integriert das PS weitere Systemkomponenten wie Platform Management Unit (PMU), Configuration Security Unit (CSU) und Battery Power Unit (BPU). Es ist in mehrere Power-Domains aufgeteilt, sodass einzelne Subsysteme auch im Low-Power-Modus betrieben werden können [32].

Application Processing Unit
Die APU des Zynq MPSoC integriert bis zu vier Arm Cortex-A53-Kerne (EG/EV), jeweils mit FPU, NEON-SIMD, Kryptographie-Einheit, MMU und getrennten 32-KB-L1-Caches. Ein gemeinsamer 1-MB-L2-Cache und eine Snoop Control Unit (SCU) gewährleisten schnelle Datenkohärenz und Kommunikation ohne externen Speicherzugriff. Die Kerne unterstützen 32-/64-Bit-Instruktionen bei bis zu 1,5 GHz Takt und ermöglichen Zugriff auf mehr als 4 GB Speicher. Über die AXI-Coherency-Extension (ACE) wird Hardware-Cachekohärenz zwischen Komponenten sichergestellt. Die NEON-Einheit beschleunigt Vektor- und Multimedia-Operationen, während die Crypto-Erweiterung AES, SHA-1/224/256, RSA und elliptische Kurven unterstützt. Systeminterrupts werden durch den externen Arm CoreLink GIC-400 nach GICv2-Standard verwaltet [32].

Real-Time Processing Unit
Die RPU des Zynq MPSoC besteht aus zwei Arm Cortex-R5-Kernen für Echtzeitanwendungen mit geringer Latenz und deterministischem Verhalten. Jeder Kern verfügt über eine FPU, eine Memory Protection Unit, getrennte 32-KB-Caches für Daten und Instruktionen sowie drei Tightly Coupled Memories (TCMs) mit insgesamt bis zu 96 KB lokalem Speicher. Die Prozessoren laufen mit bis zu 600 MHz (EV/EG) bzw. 533 MHz (CG) und können im Split Mode unabhängig oder im Lock-Step Mode redundant betrieben werden. ECC-Mechanismen (SEC-DED) ermöglichen Fehlererkennung und -korrektur. TCM-Speicher mit parallelem Zugriff sorgt für vorhersagbare Ladezeiten und deterministische Ausführung. Die RPU arbeitet im Low-Power-Bereich des Systems und bleibt auch im Low-Power-Modus aktiv. Interrupts werden durch einen dedizierten Arm CoreLink GIC-PL390 nach GICv1-Standard mit niedriger Latenz gesteuert [32].

Graphics Processing Unit
Die Zynq MPSoC integriert eine Arm Mali-400 MP2 GPU mit einem Geometrieprozessor und zwei Pixelprozessoren, jeweils mit eigener MMU und gemeinsamem 64-KB-L2-Cache. Sie unterstützt 2D- und 3D-Grafikbeschleunigung bis 667 MHz und kann grafikintensive Aufgaben über OpenGL ES 1.1/2.0 sowie OpenVG 1.0/1.1 APIs auslagern. Der L2-Cache ist über eine APB-Schnittstelle steuerbar, sodass andere Komponenten direkten Zugriff erhalten [32].

2.5.2 Programmable Logic

Die programmierbare Logik (PL) des Zynq UltraScale+ MPSoC basiert auf der 16 nm Kintex UltraScale+ FPGA-Fabric. Zentrales Element ist die Logik-Fabric, die aus zweidimensional angeordneten Configurable Logic Blocks (CLBs) besteht. Jeder CLB enthält Slices mit Lookup Tables (8 x 6 Eingänge), Flip-Flops (16) und integrierter Carry-Logik. Über programmierbare Interconnects und Switch-Matrix lassen sich diese Bausteine flexibel zu komplexen arithmetischen Schaltungen wie Addierern, Multiplikatoren oder Teilern verknüpfen. Die LUTs können dabei nicht nur als Logikgatter, sondern auch als kleiner, verteilter Speicher (Distributed RAM) genutzt werden [32].

Ergänzend zum Logik-Fabric stehen spezialisierte Speicher- und Signalverarbeitungsressourcen zur Verfügung. Block RAMs mit 36 Kb pro Tile bieten konfigurierbaren RAM-, ROM- oder FIFO-Betrieb und besitzen Dual-Port-Schnittstellen, diese lassen sich zu größeren Speicherarrays kaskadieren. UltraRAM-Blöcke mit 288 Kb, verfügbar in ausgewählten Devices, ermöglichen hochdichte und latenzarme Speicherlösungen bis zu etwa 100 Mb, ideal für datenintensive Anwendungen wie Video-Frame-Puffer. Für hochperformante arithmetische Operationen sind DSP48E2-Slices integriert, spezialisierte Multiplikator- und Signalverarbeitungseinheiten, die in Spalten angeordnet und direkt mit Block RAM verknüpft sind, um Routing-Verzögerungen zu minimieren. Durch die Möglichkeit, DSP-Slices zu kaskadieren, lassen sich zudem Funktionen mit breiteren Wortlängen realisieren [32].

2.6 PYNQ

PYNQ (Python Productivity for Zynq) ist ein von Xilinx entwickeltes Open-Source-Framework, das die Programmierung von System-on-Chip-(SoC)-Plattformen der Zynq-Familie vereinfacht. Das Framework zielt darauf ab, den Entwicklungsaufwand für hardwarebeschleunigte Anwendungen zu reduzieren. Dazu wird die Hardwarelogik des programmierbaren Logikteils (PL, Programmable Logic) über eine benutzerfreundliche Python-Schnittstelle direkt vom Prozessor (Processing System) aus gesteuert [7].

PYNQ stellt eine Linux-basierte Laufzeitumgebung bereit, in der vor allem Jupyter-Notebooks als Entwicklungs- und Testumgebung genutzt werden. Anwender können auf dieser Basis FPGA-Designs über sogenannte Overlays (vorkompilierte Bitstreams mit definierten Schnittstellen) laden und steuern. Damit lassen sich komplexe Hardwaremodule konfigurieren und in Software-Applikationen einbinden, ohne dass tiefgehendes Wissen in VHDL oder Verilog erforderlich ist.

Das Framework unterstützt die Ansteuerung von IP-Cores, AXI-Schnittstellen und Peripheriegeräten direkt über Python-Bibliotheken. Darüber hinaus bietet PYNQ eine Vielzahl vorgefertigter Bibliotheken für Anwendungen aus Bereichen wie Signalverarbeitung, Bildverarbeitung, maschinelles Lernen und Kommunikation. Die modulare Architektur erlaubt die Entwicklung eigener Overlays oder die Anpassung und Wiederverwendung bestehender Designs [1].

Aufgrund der hohen Flexibilität und der schnellen Prototypenerstellung eignet sich PYNQ besonders für Lehre, Forschung und frühe Entwicklungsphasen. Im Gegensatz zu reinem HDL-Design ermöglicht PYNQ eine deutlich verkürzte Iterationszeit, da Änderungen an der Software sofort getestet werden können und sich Hardwarefunktionen dynamisch nachladen lassen.

2.7 I2S-Protokoll

Das I2S-Protokoll ist ein serielles Busprotokoll zur Übertragung von digitalen Audiodaten zwischen Komponenten, z.B. von einem Mikrocontroller zu einem DAC oder ADC. Es überträgt Daten synchron zum Serial Clock (SCK) und verwendet zusätzlich ein Word-Select-Signal (WS), um zwischen linkem und rechtem Audiokanal zu unterscheiden [34].

Ablauf der Übertragung:

- Datenrichtung: I2S ist unidirektional; ein Master sendet, der Slave empfängt.
- Serial Clock (SCK): Takt für die seriellen Datenübertragung. Jede steigende oder fallende Flanke (abhängig von der Spezifikation) definiert ein Bit.
- Word Select (WS): Signalisiert, ob gerade der linke oder rechte Kanal übertragen wird; wechselt typischerweise nach einer festen Bitanzahl (z.B. 16 oder 24 Bit).
- Datenleitung (SD): Überträgt die Audiobits seriell. Daten werden MSB-first gesendet, meist vor der WS-Änderung.
- Master/Slave: Der Master erzeugt SCK und WS, der Slave richtet sich nach diesen Signalen.

In Abbildung 2.11 ist eine solche Übertragung dargestellt.

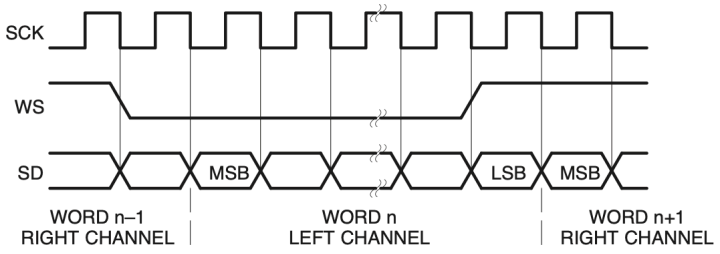

Abbildung 2.11: Ablauf des I2S-Protokolls [34]

2.8 AXI4-Stream

Mithilfe des AXI4-Stream-Interfaces lassen sich Komponenten in einer Verarbeitungskette einfach hintereinanderschalten. Jede Komponente besitzt ein Master-Interface zum Senden und ein Slave-Interface zum Empfangen von Daten. Die erste Komponente benötigt kein Slave-Interface, die letzte kein Master-Interface. Der Datentransfer erfolgt synchron zum Taktsignal ACLK [35].

Der Datenaustausch basiert auf einem 4-Phasen-Handshake[17]:

- **TVALID** (vom Master): Signalisiert dem Empfänger, dass gültige Daten auf TDATA bereitstehen.
- **TREADY** (vom Slave): Zeigt an, dass der Empfänger bereit ist, Daten zu übernehmen. Daten werden nur übertragen, wenn TVALID = 1 und TREADY = 1.
- **TDATA**: 32-Bit-Bus für die eigentlichen Daten, byteweise organisiert. Auch unterschiedlich breite Datenströme können zeitgleich übertragen werden.
- **TLAST**: Markiert das letzte Datenelement eines Pakets, z.B. für vektorielle oder komplexe Datenströme.

In Abbildung 2.12 ist eine beispielhafte AXI4-Stream-Übertragung gezeigt, bei der abwechselnd reale und imaginäre Daten übertragen werden. Die imaginären Daten sind jeweils die letzten Daten eines Pakets und werden daher über TLAST gekennzeichnet. Die Steuerung erfolgt bei steigender Taktflanke von ACLK, sodass die Datenübernahme und Signalerzeugung synchron ablaufen [35].

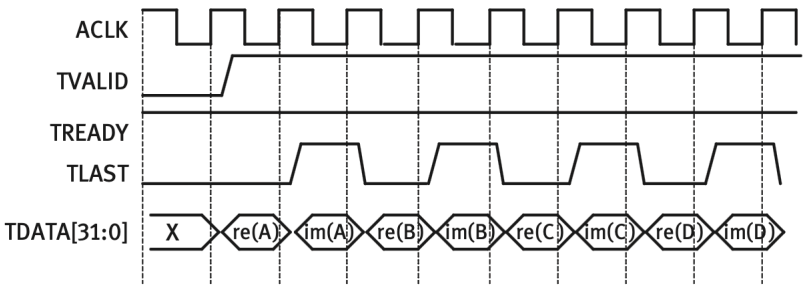

Abbildung 2.12: Ablauf der AXI4-Stream-Übertragung [35]

2.9 AXI4-Stream Video

Für die Übertragung von Bildern bietet AXI4-Stream eine spezielle Variante, das AXI4-Stream-Video, das in das zuvor beschriebene AXI4-Stream-Protokoll integriert wird. Die Synchronisation von Videodaten und die Kennzeichnung von Zeilen- und Frame-Grenzen erfolgt über das Start of Frame (SOF)-Signal, das zuvor optional als TUSER0 existierte. Es signalisiert das erste Pixel eines Videoframes oder -feldes. Der SOF-Puls ist genau eine gültige Transaktion breit und dient der Frame-Synchronisation, sodass nachgeschaltete Module den Frame korrekt erkennen und initialisieren können [11].

Am Ende jeder Zeile wird zusätzlich das End of Line (EOL)-Signal über TLAST gesendet. Es kennzeichnet das letzte Pixel einer Zeile, ist ebenfalls genau eine gültige Transaktion breit und markiert das Ende eines Scan-Lines [11].

Die Videodaten werden in Form von DATA-Vektoren übertragen, die logische Subsets der physischen Datenleitungen darstellen. Dadurch können mehrere Farbkanäle gleichzeitig in einem Datenbeat codiert werden, was die Datenübertragung effizienter gestaltet. Das Protokoll unterstützt verschiedene Farbtiefen, beispielsweise 8, 10 oder 12 Bit pro Farbkanal. Zudem werden unterschiedliche Farbräume unterstützt, darunter RGB für direkte Farbdarstellung und YUV 420 für komprimierte Videoformate. Diese Flexibilität erlaubt eine optimale Anpassung an unterschiedliche Videoauflösungen, Farbformate und Bandbreitenanforderungen, ohne dass die Protokollstruktur geändert werden muss [11].

Dieses Protokoll ermöglicht die einfache Kaskadierung von Video-Modulen und sorgt für eine deterministische Frame- und Zeilen-Synchronisation. In Abbildung 2.13 ist der Ablauf mit SOF und EOL dargestellt.

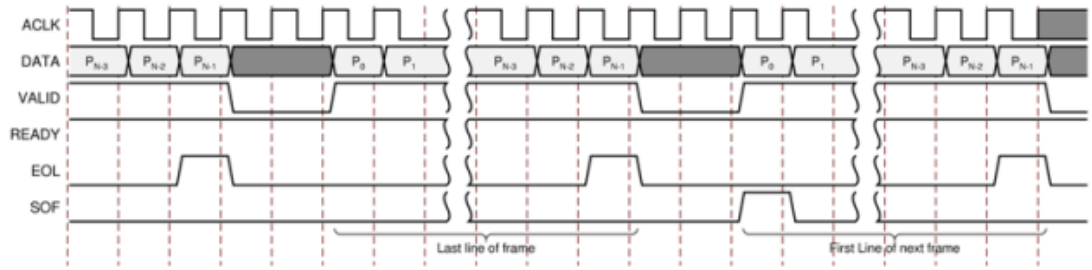

Abbildung 2.13: Ablauf der AXI4-Stream-Video-Übertragung [11]

3 Anforderungsanalyse

Das Folgende Kapitel befasst sich mit der Anforderungsanalyse. Im Rahmen der Untersuchung werden die Ausgangssituation, das Zielbild und die erforderlichen Restriktionen analysiert und erläutert.

3.1 Ausgangssituation

Diese Arbeit knüpft an eine vorangegangene Untersuchung an, in der die automatisierte Erkennung von Vogelarten anhand ihrer Gesänge erforscht wurde. Diese wurde im Rahmen eines Studierendenprojekts durchgeführt [16]. Dazu wurden über 33.000 Audioaufnahmen von Vogelarten im Raum Deutschland (mit Fokus auf Hamburg) gesammelt, vorverarbeitet und zur Klassifikation in Convolutional Neural Networks (CNNs) überführt. Die Audiodaten wurden der öffentlich zugänglichen Plattform Xeno-Canto [50] entnommen. Für die Klassifikation wurden verschiedene CNN-Modelle eingesetzt, darunter einfache Sequential-Modelle sowie EfficientNetV2. Die Validierungsgenauigkeit der Modelle variierte mit einer maximalen Präzision von 94%. Die Umsetzung erfolgte ausschließlich auf konventioneller Desktop-Hardware.

3.1.1 Funktion des Python-Codes

Für die erfolgreiche Umsetzung der bestehenden Klassifikationspipeline auf einem SoC ist ein präzises Verständnis der einzelnen Verarbeitungsschritte erforderlich. Aus diesem Grund wird im Folgenden der vorhandene Python-Code einer detaillierten Analyse unterzogen. Die vorliegende Analyse hat die Funktion, bei der Entscheidungsfindung Unterstützung zu leisten, welche Bestandteile der Verarbeitungskette für eine Hardwarebeschleunigung geeignet sind und welche Parameter zwingend beizubehalten sind.

Die einzelnen Schritte des Codes sind in der folgenden Auflistung dargestellt:

- Laden der Audiodatei
- Vorverarbeitung
- Berechnung der Mel-Spektrogramme
- Klassifikation der Mel-Spektrogramme

Laden der Audiodatei
Das System ist zur Verarbeitung von Einzeldateien im WAV-Format konzipiert. Während der Laufzeit werden die Daten geladen und verarbeitet. Die Abtastrate der Daten beträgt je nach Audiodatei 32 kHz, 44,1 kHz oder 48 kHz.

Vorverarbeitung
Die Vorverarbeitung des Audiosignals erfolgt in drei aufeinanderfolgenden Schritten:

Zunächst wird das Eingangssignal auf eine Abtastrate von 22,05 kHz gebracht. Diese Reduzierung dient der Vereinheitlichung des Datenformats und verringert gleichzeitig die Rechenlast in den nachfolgenden Verarbeitungsschritten. Um sicherzustellen, dass durch die Reduzierung der Abtastrate keine relevanten Informationen verloren gehen, wurden die spezifischen Frequenzbereiche der Vögel untersucht. Diese sind in der folgenden Auflistung dargestellt.

- Amsel (common blackbird) Frequenzbereich 1,3kHz - 8kHz [38]
- Kohlmeise (great tit) Frequenzbereich 1,5kHz - 6kHz [42]
- Mönchsgrasmücke (eurasian blackcap) Frequenzbereich 2,1kHz - 4,4kHz [44]
- Zilp Zalp (common chiffchaff) Frequenzbereich 3kHz - 7kHz [48]
- Rotkehlchen (european robin) Frequenzbereich 1kHz - 8kHz [41]
- Singdrossel (song trush) Frequenzbereich 1kHz - 6kHz [45]
- Buchfink (common chaffinch) Frequenzbereich 2,8kHz - 6,5kHz [40]
- Zaunkönig (eurasian wren) Frequenzbereich 2kHz - 9kHz [46]
- Blaumeise (eurasian blue tit) Frequenzbereich 4kHz - 9kHz [43]
- Buntspecht (gereat spotted woodpecker) Frequenzbereich 500Hz - 3kHz [47]

Die Frequenzen liegen in einem Bereich von 500 Hz bis 9 kHz und damit deutlich unterhalb der Nyquist-Frequenz von Fs/2, bei 11,025 kHz.

Im nächsten Schritt werden störende Hintergrundgeräusche entfernt. Hierzu kommt die Funktion *reduce_noise()* aus dem Modul *noisereduce* zum Einsatz. Dabei wird das Audiosignal mittels Short-Time Fourier Transform (STFT) in ein komplexes Spektrogramm überführt. Auf Basis der spektralen Energie in leisen Abschnitten wird ein Rauschprofil abgeleitet und vom Spektrum subtrahiert. Anschließend erfolgt die Rücktransformation des bereinigten Spektrums in ein Zeitsignal mittels inverser STFT (ISTFT).

Im letzten Verarbeitungsschritt wird stillehaltiges Material entfernt, um den Fokus auf relevante Signalbereiche zu legen. Hierfür wird die Funktion *split_on_silence()* aus dem Modul *pydub* verwendet. Diese segmentiert das Signal anhand von definierten Schwellenwerten für Lautstärke und Mindestdauer der Stille. Die entsprechenden Parameter wurden experimentell für jede Zielklasse individuell angepasst, um eine möglichst präzise Trennung zu erreichen.

Berechnung der Mel-Spektrogramme
Die Erzeugung des Mel-Spektrogramms erfolgt analog zuden Ausführungen in Kapitel 2.3 unter Verwendung der Funktion *librosa.feature.melspectrogram()* [4]. Im Rahmen der implementierten Verarbeitungskette wurden dabei die folgenden Parameter verwendet:

- **Spektrogrammlänge**: 4 Sekunden
- **Fensterfunktion**: Hanning-Fenster
- **Überlappung**: 50 %
- **FFT-Punkte**: 2048
- **Mel-Filter**: 128

Nach der Berechnung des Mel-Spektrogramms wird die Amplitude mittels logarithmischer Skalierung in ein Dezibel-Spektrogramm überführt. Abschließend erfolgt eine Normalisierung des Spektrogramms sowie eine Skalierung auf eine Bildgröße von 224 x 224 Pixeln, um es für die nachfolgende Klassifikation nutzbar zu machen.

Klassifikation der Mel-Spektrogramme
Das im Studierendenprojekt [16] trainierte CNN wurde mit TensorFlow/Keras implementiert und besteht aus *Convolution*-, *MaxPooling*-, *Flatten*- und *Dense*-Schichten. Eingabe ist ein 224x224 Pixel großes Mel-Spektrogramm, Ausgabe ist ein Vektor mit zehn Wahrscheinlichkeitswerten für die Zielklassen.

Abbildung 3.1 zeigt den strukturellen Aufbau des Netzwerks einschließlich der Zwischengrößen der Daten in den jeweiligen Schichten.

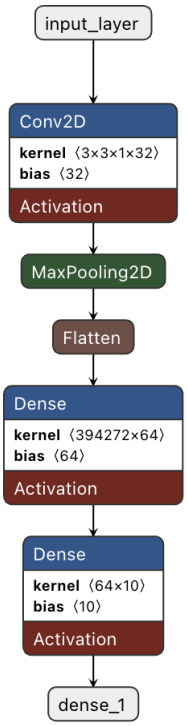

Abbildung 3.1: Architektur des CNN

3.1.2 Aktuelle Performance

Im Rahmen der Analyse des zugrunde liegenden Studierendenprojekts zur Portierung auf ein SoC kommt den Performancekennzahlen – insbesondere den Laufzeiten der einzelnen Verarbeitungsschritte sowie der Klassifikationsgenauigkeit – eine zentrale Bedeutung zu. Sie dienen als maßgebliche Referenz, um zu einem späteren Zeitpunkt einen fundierten Vergleich mit dem Zielsystem durchführen zu können.

Laufzeit
Die Laufzeitanalyse wurde mithilfe der datetime-Funktion in Python durchgeführt, wobei jeweils die Differenz zwischen Start- und Endzeitpunkt der gesamten Verarbeitung gemessen wurde. Als Testplattform kam ein Apple M2 Pro Prozessor zum Einsatz. Die Messungen wurden mehrfach ausgeführt, um Schwankungen zu erfassen; dokumentiert wurden jeweils die minimalen und maximalen Laufzeiten.

Zu beachten ist, dass die Vorverarbeitung nicht isoliert messbar ist, da sie funktional in nachfolgende Verarbeitungsschritte integriert ist. Eine getrennte zeitliche Erfassung der einzelnen Module war daher im Rahmen dieser Messung nicht möglich.

- Audiosample Laden: 3-8 ms
- Vorverarbeitung: nicht seperat messbar
- Mel-Spektrogramm: 10-30 ms
- Klassifikation: 45-70 ms

Insgesamt beträgt die Verarbeitungszeit pro 4-Sekunden-Audioausschnitt etwa 58 ms bis 108 ms. Das Audiosignal liegt dabei vollständig zu Beginn der Verarbeitung vor. Die Messung umfasst sämtliche Schritte - von der Erzeugung des Mel-Spektrogramms bis zur Ausgabe der Klassifikationsergebnisse.

Leistungsmetriken
Das im Rahmen des Studierendenprojekts entwickelte Convolutional Neural Network (CNN) erzielt auf dem Validierungsdatensatz eine Klassifikationsgenauigkeit von 94 %. Zur weiterführenden Bewertung des Modells wurde ergänzend eine Konfusionsmatrix erstellt (siehe Abbildung 3.2).

Die Abbildung 3.2 veranschaulicht die Verteilung korrekt und fehlerhaft klassifizierter Instanzen über sämtliche Zielklassen hinweg. Auffällig ist dabei die nahezu perfekte Ausprägung der Hauptdiagonale. Dies deutet auf eine sehr zuverlässige Unterscheidung der Kategorien hin. Dies ermöglicht nicht nur eine Einschätzung der Gesamtleistung, sondern auch eine gezielte Analyse potenzieller Schwächen – etwa im Hinblick auf vereinzelte Verwechslungen akustisch ähnlicher Klassen. Solche Erkenntnisse sind insbesondere für eine spätere Implementierung relevant, da sie Rückschlüsse auf die Robustheit des Modells im praktischen Einsatz erlauben und gezielte Optimierungen zur weiteren Verbesserung der Systemzuverlässigkeit ermöglichen.

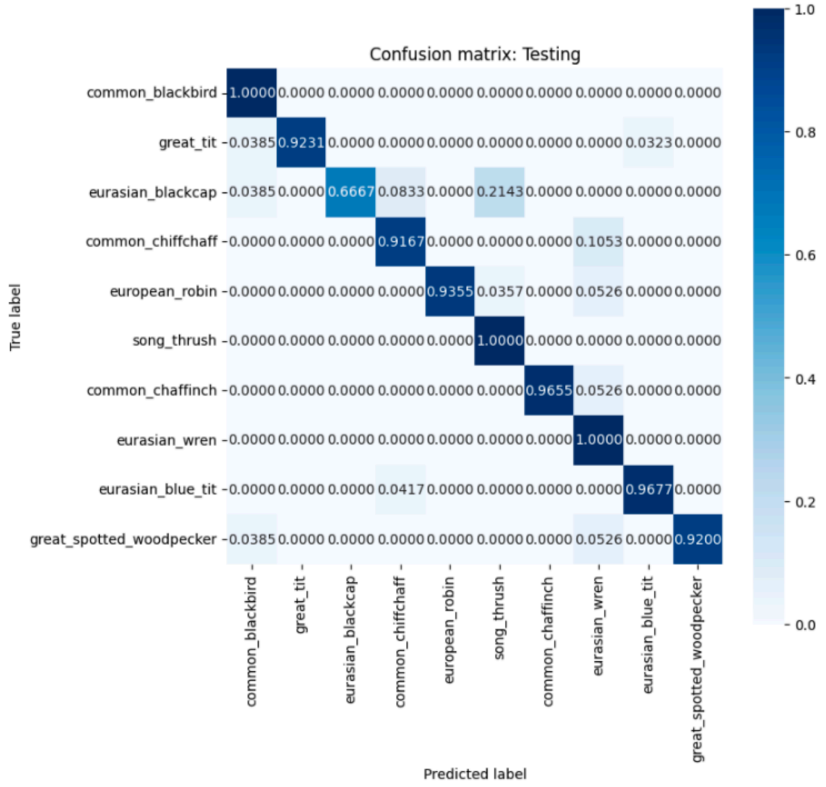

Abbildung 3.2: Confusion Matrix der Klassifikation

3.2 Zielbild

Aufbauend auf dem zuvor in Abschnitt 3.1 analysierten Studierendenprojekt besteht das zentrale Ziel dieser Arbeit darin, die bestehende, auf Python basierende Klassifikationspipeline auf eine SoC-Plattform zu übertragen. Das Zielsystem zielt darauf ab, eine echtzeitfähige Verarbeitung akustischer Signale zu ermöglichen. Die Realisierung dieser Anforderung bedingt die Abbildung sämtlicher essentieller Schritte auf der SoC-Plattform. Diese Schritte umfassen die Signalaufnahme, die Vorverarbeitung, die Merkmalsextraktion sowie die Klassifikation durch ein CNN. Im Vordergrund steht dabei eine effiziente Umsetzung mit geringer Latenz, die für den Einsatz in eingebetteten Systemen geeignet ist. Die Ausgabe der Klassifikationsergebnisse soll über ein Display erfolgen.

3.3 Anforderungsspezifikation

Für die Überführung der in Abschnitt 3.1 beschriebenen Ausgangslage in das in Abschnitt 3.2 dargestellte Zielbild sowie für eine fundierte Bewertung der Portierung ist eine präzise Definition der Anforderungen notwendig. Auf Grundlage der vorangegangenen Analyse werden daher spezifische Anforderungen abgeleitet, die sich in Methodische Anforderungen, Anforderungen an die Hardware und an die Datenverarbeitung gliedern.

3.3.1 Methodische Anforderungen

Bei der Entwicklung des Systems wird ein Hardware-Software-Co-Design-Ansatz verfolgt. Das Ziel besteht darin, die funktionalen Aufgaben des Gesamtsystems systematisch zwischen Softwarekomponenten (die beispielsweise auf Prozessoren ausgeführt werden) und Hardwarekomponenten (wie FPGA-basierte Beschleuniger) aufzuteilen. Durch diese Vorgehensweise sollen folgende methodische Anforderungen erfüllt werden:

Strukturierte Partitionierung:
Frühzeitige Analyse und Aufteilung der Algorithmen in Hardware- und Softwareanteile, basierend auf Kriterien wie Latenz, Ressourceneinsatz und Flexibilität.

Vergleich unterschiedlicher Implementierungsmethoden:
Um fundierte Entscheidungen über den optimalen Architekturansatz zu treffen, werden für ausgewählte Module alternative Implementierungsstrategien untersucht.

Experimenteller Einsatz moderner Werkzeuge:
Es soll gezielt mit High-Level Synthesis (HLS) und weiteren Entwicklungsumgebungen experimentiert werden, um den Einfluss unterschiedlicher Abstraktionsebenen und Toolflows auf Entwicklungsaufwand, Performanz und Ressourceneffizienz zu bewerten.

Die methodische Vorgehensweise legt den Schwerpunkt nicht nur auf ein optimales Endergebnis, sondern auch auf die Identifizierung von Stärken und Schwächen verschiedener Designmethoden. Damit soll eine belastbare Grundlage geschaffen werden, um die Eignung von HLS gegenüber klassischen HDL-Entwurfsansätzen kritisch einschätzen und praxisrelevante Empfehlungen für zukünftige Projekte ableiten zu können.

3.3.2 Anforderungen an die Datenverarbeitung

Die Anforderungen an die Datenverarbeitung basieren auf der in Abschnitt 3.1.1 durchgeführten Analyse der Funktion des Python-Codes. Die einzelnen Verarbeitungsschritte wurden dort bereits erläutert. Im Folgenden werden sie als formale Anforderungen zusammengefasst und festgehalten, um eine eindeutige Identifizierung und Umsetzung im aktuellen Projekt zu ermöglichen.

- **DV-1**: Um die Kompatibilität mit der Referenzimplementierung in Python sicherzustellen, müssen Audiosignale mit einer Abtastrate von 22,05 kHz einlesen werden.
- **DV-2**: Die kontinuierlichen Audiodaten müssen in Fenster mit jeweils 2048 Werten segmentiert werden.
- **DV-3**: Die Datenfenster müssen sich zu 50 % überlappen.
- **DV-4**: Jedes Datenfenster muss mit einem Hanning-Fenster multipliziert werden.
- **DV-5**: Die FFT muss mit einer Größe von 2048 Punkten durchgeführt werden.
- **DV-6**: Für die Spektralanalyse müssen 128 Mel-Filter verwendet werden.
- **DV-7**: Die Amplituden der Mel-Spektren müssen logarithmisch skaliert werden.
- **DV-8**: Die Mel-Spektrogramme müssen auf eine Auflösung von 224 x 224 Pixel skaliert werden.
- **DV-9**: Die Mel-Spektrogramme müssen kontinuierlich und fortlaufend ohne Signalverlust zur Klassifizierung bereitgestellt werden.
- **DV-10**: Die Ausgabe der Wahrscheinlichkeiten der zehn Klassen erfolgt über ein Display.

3.3.3 Anforderungen an die Hardware

Basierend auf dem Zielbild, den Methodischen Anforderungen und den Anforderungen an die Datenverarbeitung wurden die Anforderungen an die Hardware festgelegt, um eine bestmögliche Umsetzung zu gewährleisten.

- **H-1** Das Zielsystem *muss* ein System-on-a-Chip (SoC) sein, das mindestens einen fest integrierten Prozessor (Hardcore-CPU) und ein FPGA auf einem gemeinsamen Chip enthält.
- **H-2** Die Kommunikation zwischen Prozessor und FPGA *muss* über ein Hochgeschwindigkeitsinterface wie AXI, PCIe oder eine vergleichbare Technologie erfolgen, um eine Datenübertragung mit minimaler Latenz sicherzustellen.
- **H-3** Das System *muss* über ausreichend Speicherressourcen verfügen, um sowohl den Programmcode als auch die Datenpuffer effizient verarbeiten zu können. Das verwendete CNN-Modell beansprucht bereits ca. 302 MB Speicherplatz. Für zukünftige Erweiterungen, beispielsweise durch den Einsatz eines Betriebssystems wie PetaLinux, ist zusätzlicher Arbeitsspeicher einzuplanen (mindestens 512 MB bei Zynq-7000-SoCs bzw. 2 GB bei Zynq UltraScale+ [13]). Daher *muss* das SoC mindestens 2 GB RAM bereitstellen, besser jedoch mehr, um ausreichende Reserven für spätere Anforderungen zu gewährleisten.
- **H-4** Das FPGA *muss* mindestens 500 DSP-Slices und 50.000 Logikzellen bereitstellen, um eine Auslagerung rechenintensiver Operationen zu ermöglichen.
- **H-5** Das Audiointerface *muss* eine Abtastrate von 22,05 kHz oder ein ganzzahliges Vielfaches davon unterstützen, sodass eine verlustfreie Reduktion auf 22,05 kHz mittels einfachem Downsampling möglich ist.
- **H-6** Die Kosten des Zielsystems *sollen* so gering wie möglich sein, ohne die funktionalen und leistungsbezogenen Anforderungen (H-1 bis H-5) zu verletzen.
- **H-7** Das System *muss* über ein integriertes Display verfügen, das folgende Eigenschaften erfüllt:Unterstützung einer Standardauflösung (mindestens 800 x 480 Pixel), ausreichend kompakte Abmessungen, um das Gesamtsystem handlich zu halten (ca. 7 Zoll), direkte Ansteuerbarkeit über gängige Schnittstellen (z. B. HDMI oder MIPI-DSI), eine Aktualisierungsrate, die eine flüssige Darstellung der Klassifikationsergebnisse ermöglicht.
- **H-8** Alle Hardwarekomponenten *müssen* in einem Gehäuse integriert werden, das Schutz vor äußeren Einflüssen bietet und einen mobilen bzw. stationären Einsatz zu Testzwecken ermöglicht.

4 Konzept und Systemübersicht

Im folgenden Kapitel werden der Aufbau des Systems sowie die zugrunde liegenden Konzepte im Detail erläutert. Die Bewertung erfolgt dabei auf Basis der in Kapitel 3.3 definierten Anforderungen.

4.1 Systemaufbau

Die Planung des Systemaufbaus orientiert sich an der in Abschnitt 3.2 dargestellten Beschreibung des Zielbilds. Wie in Abbildung 4.1 dargestellt, erfolgt zunächst der Empfang der Audiodaten über einen Audioschnittstelle. Anschließend werden die Audiodaten zur Verarbeitung an ein SoC übertragen. Auf diesem findet die Datenverarbeitung und klassifikation statt. Die resultierende Klassifikation in eine von zehn Klassen wird über ein Display an den Nutzer ausgegeben.

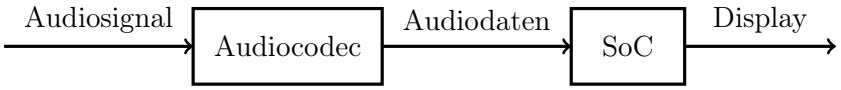

Abbildung 4.1: Systemaufbau als Blockdiagramm

4.2 Hardware Konzept

Die Auswahl der Hardware erfolgt in einem strukturierten Prozess. Zunächst wird das zentrale System-on-Chip (SoC) anhand der Anforderungen H-1 bis H-6 bestimmt. Diese Vorgaben definieren die grundlegende Architektur – Prozessor-FPGA-Integration, Speicherressourcen, Schnittstellen und Rechenkapazitäten. Auf dieser Basis kommen ausschließlich Development Boards mit einem Zynq UltraScale+ SoC von AMD/Xilinx infrage. Wie in Kapitel 2.5 beschrieben, vereint diese Plattform ARM-Prozessoren mit programmierbarer Logik, womit die Anforderungen H-1 und H-2 direkt erfüllt werden.

Darüber hinaus ist die Entwicklungsumgebung mit Vivado und Vitis bereits bekannt, was den Implementierungsaufwand reduziert und Entwicklungsrisiken minimiert.

Aufbauend auf der SoC-Auswahl werden im nächsten Schritt die Peripheriekomponenten spezifiziert. Hierzu zählen insbesondere das Display (H-7), das eine standardisierte Auflösung bei kompakter Baugröße bieten muss, sowie das Gehäuse (H-8), das alle Systemkomponenten integriert und ein robustes Gesamtsystem sicherstellt. Auf diese Weise werden nicht nur die Rechenplattform, sondern auch Bedien- und Integrationseigenschaften konsistent aufeinander abgestimmt.

4.2.1 ZCU104

Das ZCU104 Evaluation Board (siehe Abbildung 4.2) stellt ein leistungsfähiges Entwicklungsboard auf Basis des Zynq UltraScale+ MPSoC ZU7EV dar. Es wurde für professionelle Anwendungen mit hohen Anforderungen an Rechenleistung, FPGA-Fabric und Videoverarbeitung konzipiert.

Abbildung 4.2: ZCU104 [15]

Bewertung der Anforderungen:

- H-3: Das Speicherangebot wird aufgrund der vorhandenen Kapazität von 2 GB DDR4 als ausreichend bewertet.
- H-4: Es werden 1.728 DSP-Slices und 504k Systemlogikzellen bereitgestellt, wodurch eine Beschleunigung komplexer Operationen auf der Hardwareebene ermöglicht wird.

- H-5: Ein dediziertes Audiointerface mit einer Frequenz von 44,1 kHz ist nicht vorhanden, kann jedoch über FMC-Erweiterungen oder PMOD, USB etc. nachgerüstet werden.
- H-6: Die Kosten belaufen sich auf 1.906,84 Euro und sind somit im oberen Bereich anzusiedeln. Diese ist jedoch durch die hohe Leistung des Geräts zu rechtfertigen.

4.2.2 Genesys ZU

Das Genesys ZU Entwicklungsboard von Digilent (siehe Abbildung 4.3) stellt eine solide Plattform für Forschung, Lehre und Prototyping bereit. Die Basis bildet ein Zynq UltraScale+ ZU5EV SoC.

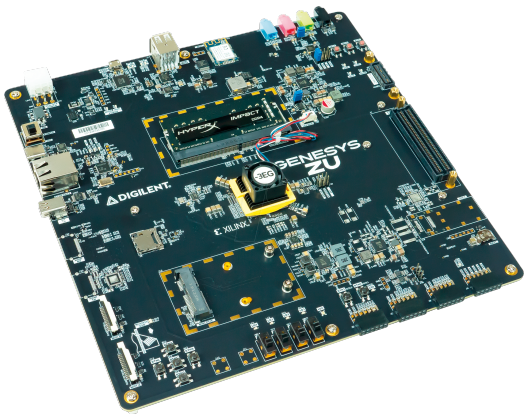

Abbildung 4.3: Genesys ZU [24]

Bewertung der Anforderungen:

- H-3: Das Speicherangebot wird aufgrund der vorhandenen Kapazität von 4 GB DDR4 als ausreichend bewertet.
- H-4: Es werden 1.248 DSP-Slices und 256k Systemlogikzellen bereitgestellt, wodurch eine Beschleunigung komplexer Operationen auf der Hardwareebene ermöglicht wird.
- H-5: Es wurde ein dediziertes Audiointerface bereitgestellt, das eine Abtastrate von 44,1 kHz aufweist. Das vorliegende Audiointerface verwendet den Audiocodec ÄDAU1761".

- H-6: Gemäß der Preisgestaltung beläuft sich der Preis des Genesys ZU auf 1.895,25 Euro. Demzufolge ist das Produkt im oberen Preissegment einzuordnen.

4.2.3 Kria KV260

Das Kria KV260 Vision AI Starter Kit (siehe Abbildung 4.4) stellt eine modulare Plattform für KI- und Vision-Anwendungen dar. Wie bereits dargelegt, findet ein Zynq UltraScale+ SoC Anwendung, wobei der Aufbau kompakter gestaltet ist und eine Optimierung für den Einsatz in Edge-Systemen erfolgt ist.

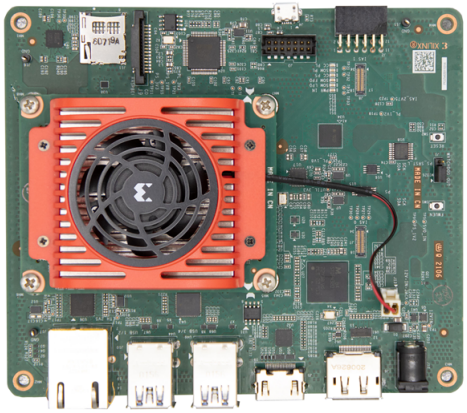

Abbildung 4.4: Kria KV260 [3]

Bewertung der Anforderungen:

- H-3: Das Speicherangebot wird aufgrund der vorhandenen Kapazität von 4 GB DDR4 als ausreichend bewertet.
- H-4: Es werden 1.248 DSP-Slices und 256k Systemlogikzellen bereitgestellt, wodurch eine Beschleunigung komplexer Operationen auf der Hardwareebene ermöglicht wird.
- H-5: Es wurde festgestellt, dass kein integriertes Audiointerface mit einer Frequenz von 44,1 kHz vorhanden ist. Die Realisierung ist ausschließlich über externe Schnittstellen (USB, PMOD) möglich.

- H-6: Das Kria KV260 ist ein kostengünstiges Board, dessen Preis bei rund 292,14 Euro liegt. Es eignet sich daher ideal für preisbewusste Projekte.

4.2.4 PMOD I2S2 Modul

Da weder das Kria KV260 noch das ZCU104 über ein integriertes Audiointerface verfügen, muss eine externe Lösung zur Audioeingabe angebunden werden. Das PMOD I2S2-Modul von Digilent (siehe Abbildung 4.5) stellt hierfür eine kompakte und unkomplizierte Erweiterung dar. Es ermöglicht sowohl die Aufnahme als auch die Wiedergabe von Audiosignalen über das standardisierte I2S-Protokoll und ist damit ideal geeignet, FPGA- oder SoC-basierte Systeme um Audiofunktionen zu ergänzen. Durch die Unterstützung einer Abtastrate von 44,1 kHz wird die Anforderung H-5 vollständig erfüllt. Zusätzlich überzeugt das Modul durch seine Wirtschaftlichkeit, die sich in einem Preis von lediglich 27,78 Euro widerspiegelt.

Abbildung 4.5: PMOD I2S2 [5]

4.2.5 Auswahl

Die betrachteten Boards unterscheiden sich in ihrer Ausstattung und ihren Kosten, erfüllen aber alle grundlegenden funktionalen Anforderungen. Tabelle 4.1 fasst die für die Auswahl des Zielsystems wesentlichen Merkmale zusammen. Alle Boards erfüllen die

Tabelle 4.1: Vergleich der Anforderungen der Hardware

Anforderung	ZCU104	Genesys ZU	Kria KV260
H-1	✓	✓	✓
H-2	✓	✓	✓
H-3	2GB DDR4	4GB DDR4	4GB DDR4
H-4	1.728	1.248	1.248
H-5	PMOD I2S2	Intern	PMOD I2S2
H-6	1.906,84€	1.895,25€	292,14€

Anforderungen H-1 bis H-4 und lassen sich mithilfe des PMOD I2S2 oder integrierter Schnittstellen auch für Audioanwendungen gemäß H-5 nutzen. Ausschlaggebend für die endgültige Auswahl war daher die Anforderung H-6, die Kosten. In diesem Punkt bietet das Kria KV260 bei vergleichbarer Funktionalität einen erheblichen Preisvorteil gegenüber den anderen Boards und wurde daher als Zielplattform gewählt.

4.2.6 Display

Für die Ausgabe der Klassifikationsergebnisse wird ein Display benötigt, das direkt über HDMI betrieben werden kann. Das verwendete Development Board Kria KV260 verfügt über einen entsprechenden HDMI-Ausgang, sodass keine zusätzlichen Schnittstellen oder Adapter erforderlich sind. Es wird ein handelsübliches Display mit Standardauflösung ausgewählt, wie es auf gängigen Online-Plattformen verfügbar ist. Damit werden die in Anforderung H-7 definierten Kriterien an Auflösung, Schnittstelle und Kompaktheit erfüllt.

4.2.7 Gehäuse

Für die Integration aller Systemkomponenten wird ein Gehäuse benötigt, das Schutz, Stabilität und Kompaktheit gewährleistet. Um diese Anforderungen effizient umzusetzen, wird das Gehäuse mittels 3D-Druck gefertigt. Das Design erfolgt in der bekannten CAD-Software Fusion360, wodurch eine präzise Anpassung an die Abmessungen des Kria KV260, des Displays und der weiteren Peripheriekomponenten möglich ist. Diese Vorgehensweise erlaubt eine kosteneffiziente und flexible Umsetzung sowie eine einfache Modifikation bei zukünftigen Anpassungen und erfüllt damit die Anforderungen aus H-8.

4.3 Methodisches Konzept

Die Datenverarbeitung wird auf Basis eines Hardware-Software-Co-Design-Ansatzes entwickelt. Dabei werden Algorithmen und Funktionen gezielt auf Hardware- oder Softwarekomponenten verteilt. Die Partitionierung erfolgt nicht statisch, sondern wird im Rahmen mehrerer Implementierungsvarianten untersucht.

Für die Umsetzung werden ausschließlich AMD/Xilinx-Tools (Vivado, Vitis, Vitis-HLS, Vitis-AI, PYNQ) verwendet. Dies ergibt sich daraus, dass ein AMD/Xilinx-SoC als Zielplattform genutzt wird und die entsprechenden Toolchains bereits aus dem Hochschulumfeld bekannt sind. Durch diese Vorkenntnisse kann der Entwicklungsprozess effizienter gestaltet und eine sichere Handhabung der Werkzeuge gewährleistet werden.

Es ist vorgesehen, mehrere zentrale Systemmodule mit unterschiedlichen Methoden zu implementieren, zu synthetisieren und anschließend hinsichtlich Performance, Ressourcenverbrauch und Entwicklungsaufwand zu vergleichen. Ziel ist es, die Vor- und Nachteile der verschiedenen Designstrategien klar herauszuarbeiten und fundierte Empfehlungen für ein optimales Hardware-Software-Co-Design in Verbindung mit HLS abzuleiten.

Um dieses Ziel zu erreichen, muss die Kommunikation zwischen den Modulen vereinheitlicht werden. Ein hierfür weit verbreitetes Schnittstellenprotokoll auf SoCs ist AXI-Stream. Es ermöglicht eine standardisierte und modulare Anbindung von Hardware- und Softwarekomponenten. Bei der Implementierung ist zu berücksichtigen, dass die Busbreite der für den Datentransfer eingesetzten DMA-Schnittstellen eine Zweierpotenz sein muss [10].

4.4 Aufbau der Datenverarbeitung

Die Datenverarbeitung bildet das zentrale Element dieser Arbeit und umfasst sämtliche in den Anforderungen DV-1 bis DV-10 definierten Verarbeitungsschritte. Um eine effiziente und strukturierte Umsetzung zu gewährleisten, wird die gesamte Verarbeitungskette in mehrere funktionale Untermodule gegliedert, die sich am zeitlichen Ablauf der Verarbeitung orientieren. Diese Modularisierung erleichtert sowohl die Implementierung als auch eine gezielte Optimierung einzelner Prozessschritte.

Der Datenfluss beginnt mit der Erfassung der Audiodaten über das in Abschnitt 4.2.4 beschriebene I2S2-PMOD-Modul. Dieses verwendet das I2S-Protokoll und erfordert entsprechende Taktsignale. Die ankommenden Daten liegen seriell vor und müssen für die weitere Verarbeitung parallelisiert werden. Die Samplingrate beträgt 44,1 kHz. Da dieser Schritt kontinuierlich ausgeführt wird, wird er in der Audio Input Unit (AIU) zusammengefasst.

Die Samplingrate ist mit 44,1 kHz zu hoch. So wird das Eingangssignal zunächst auf eine Samplingrate von 22,05 kHz reduziert (DV-1), anschließend in Blöcke mit 2048 Punkten segmentiert (DV-2) und mit einer Überlappung von 50 % versehen (DV-3). Außerdem wird gemäß DV-4 ein Hanning-Fenster auf jedes Segment angewendet, wodurch sich die Verarbeitungsfrequenz auf rund 21,53 Hz reduziert. Diese Stufe wird Signal Framing Unit (SFU) genannt.

Im Anschluss erfolgt eine Frequenzanalyse mittels einer 2048-Punkte-FFT (DV-5), gefolgt von der Überführung in Mel-Frequenzen mithilfe von 128 Mel-Filtern (DV-6). Die daraus resultierenden Amplitudenwerte werden logarithmiert (DV-7). Mehrere solcher Frames werden gemäß DV-8 zu einem Mel-Spektrum der Größe 224 x 224 Pixel zusammengefügt. Diese Verarbeitungseinheit wird als Mel Processing Unit (MPU) bezeichnet. Die Ausgabe erfolgt unter Berücksichtigung der für die Klassifikation erforderlichen Verarbeitungsgeschwindigkeit (DV-9), die bei der Implementierung ermittelt wird.

Das erzeugte Mel-Spektrum wird anschließend klassifiziert. Die dabei entstehenden Wahrscheinlichkeiten für die zehn möglichen Klassen werden gemäß DV-10 ausgegeben. Für diese Aufgabe ist die Audio Classification Unit (ACU) zuständig.

Ein schematischer Überblick über die Datenflüsse zwischen den Modulen ist in Abbildung 4.6 dargestellt. Im weiteren Verlauf werden die einzelnen Module detailliert beschrieben und hinsichtlich ihrer funktionalen Umsetzung analysiert.

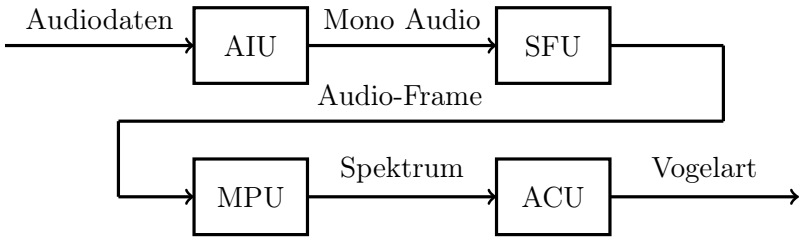

Abbildung 4.6: Blockdiagramm der Datenverarbeitung

4.4.1 Audio Input Unit

Wie in Abschnitt 4.2.5 dargelegt, erfolgt der Empfang der Audiodaten über den PMOD I2S2-Adapter, wobei die Übertragung mithilfe des I2S-Protokolls realisiert wird. Die Daten liegen dabei in 24-Bit-Auflösung innerhalb eines 32-Bit-Frames vor, wobei die Übertragung der rechten und linken Kanäle getrennt erfolgt. In Anlehnung an gängige Audio-

klassifikationsverfahren wird in diesem Projekt Mono-Audio verwendet, sodass lediglich der linke Kanal in die weitere Verarbeitung einfließt. Die empfangenen Audiodaten sollen als 16-Bit AXI-Stream ausgeben werden.

Die Umsetung einer solchen Schnittstelle erfolgt üblicherweise in einer Hardwarebeschreibungssprache wie VHDL. Alternativ kann sie auch mithilfe von HLS realisiert werden. Entsprechend dem methodischen Konzept aus Abschnitt 4.3 sollen daher zwei Varianten umgesetzt: eine klassische Umsetzung in VHDL sowie eine HLS-basierte Umsetzung, um einen direkten Vergleich hinsichtlich Entwicklungsaufwand und Effizienz zu ermöglichen.

4.4.2 Signal Framing Unit

Die Signal Framing Unit ist ein zentrales Modul innerhalb der Audioverarbeitungskette. Sie übernimmt die Vorverarbeitung der Audiodaten und gliedert sich in drei aufeinanderfolgende Teilschritte:

- Abtastratenumsetzung von 44,1 kHz auf 22,05 kHz unter Einsatz eines geeigneten FIR-Antialiasing-Filters. (DV-1)
- Framing der kontinuierlichen Audiodaten in Fenster mit einer Länge von 2048 Werten mit 50 % Überlappung. (DV-2 und DV-3)
- Fensterung der Segmente mit einem Hanning-Fenster. (DV-4)

Die Umsetzung der Abtastratenanpassung erfordert ein Antialiasing-Filter, um Aliasing-Effekte beim Downsampling auf die in Vorgabe DV-1 geforderten 22,05 kHz zuverlässig zu vermeiden. Die Auslegung des Filters stützt sich auf veröffentlichte Analysen zur spektralen Verteilung von Vogelgesängen (siehe Abschnitt 3.1.1), nach denen die relevanten Frequenzen typischerweise im Bereich von 500 Hz bis 9 kHz liegen. Auf dieser Grundlage wurde die obere Grenzfrequenz des Antialiasing-Filters definiert. Die erforderliche Filterordnung lässt sich näherungsweise mit folgender Formel bestimmen:

$$N_{FIR} \approx \frac{2}{3} log_{10}\left(\frac{1}{10 \cdot \delta_{pass} \cdot \delta_{stop}}\right) \cdot \left(\frac{F_s}{f_{stop} - f_{pass}}\right) \tag{4.1}$$

Obwohl die obere Grenzfrequenz durch die Analyse prinzipiell vorgegeben ist, beeinflussen sowohl die exakte Wahl der Übergangsfrequenz als auch die Dämpfung im Sperrband maßgeblich die Filterordnung und damit den Ressourcenbedarf auf dem FPGA. Um ein optimales Verhältnis zwischen spektraler Qualität und Hardwareaufwand zu finden, wurden verschiedene Grenzfrequenzen und Sperrbanddämpfungen systematisch untersucht und in Tabelle 4.2 gegenübergestellt.

Tabelle 4.2: Vergleich der Filterordnung

f_{pass} / Dämpfung	20 dB	40 dB	60 dB	80 dB
8,5 kHz	23	35	47	58
8,75 kHz	26	39	52	65
9 kHz	30	44	58	73
9,25 kHz	34	50	67	83
9,5 kHz	39	58	78	97

Die endgültige Auswahl der Filterordnung erfolgt erst in der Implementierungsphase und richtet sich nach den verfügbaren Ressourcen sowie den erreichbaren Latenz- und Leistungswerten. Gemäß der Methodik des HW-SW-Co-Designs wird die SFU in zwei Varianten realisiert: einerseits als reine Softwareimplementierung auf dem ARM-Core, andererseits auf Basis desselben Codes unter Verwendung von HLS zur Umsetzung in Hardware. Durch diesen Vergleich lässt sich bewerten, in welchem Umfang sich eine Hardwarebeschleunigung hinsichtlich Laufzeit und Entwicklungsaufwand gegenüber einer reinen Softwarelösung lohnt.

4.4.3 Mel Processing Unit

Die Aufgabe der Mel Processing Unit (MPU) besteht darin, aus den zuvor gefensterten Audiodaten das Mel-Spektrum zu berechnen. In mehreren wissenschaftlichen Arbeiten (vgl. [18, 8, 31]) wurde der MFCC-Algorithmus auf FPGA-Plattformen implementiert, wobei die Berechnung des Mel-Spektrums stets einen zentralen Zwischenschritt darstellt. Diese Arbeiten dienen als konzeptionelle Grundlage für die MPU. Der grundlegende Ablauf entspricht der Darstellung in Kapitel 2.3.

Die Studien zeigen außerdem, dass die einzelnen Verarbeitungsschritte rechenintensiv sind und sich daher besonders für eine Hardwareimplementierung eignen. Aus methodischer Sicht bietet dieses Modul somit eine geeignete Basis, um die Effizienz unterschiedli-

cher HLS-Implementierungen systematisch zu untersuchen. Hierzu wird einerseits die in Kapitel 2.3 beschriebene Standardvariante realisiert, andererseits eine optimierte Version, deren Konzeption im Folgenden beschrieben wird.

In Arbeit [25] wurde gezeigt, dass bei der Berechnung eines Mel-Spektrums insbesondere zwei Verarbeitungsschritte den größten Rechenaufwand verursachen: die FFT sowie die Anwendung der Mel-Filterbank. Aus diesem Grund werden für die optimierte Variante der MPU alternative Implementierungen dieser beiden Teilmodule untersucht, um deren Effizienz gezielt zu verbessern.

RFFT

Da es sich bei Audiosignalen um rein reelle Eingangsdaten handelt, ist die Verwendung einer herkömmlichen FFT-Implementierung, die auf komplexwertige Signale ausgelegt ist, nicht optimal. In der Praxis existieren zwei gängige Ansätze, um diesen Umstand effizient zu berücksichtigen:

- **Zero Padding des Imaginärteils:**
  Hierbei werden die imaginären Anteile der Eingangsdaten künstlich mit Nullen aufgefüllt. Dadurch ist die Verwendung einer konventionellen komplexen FFT-Implementierung ohne strukturelle Änderungen möglich. Allerdings führt dieses Verfahren zu einer ineffizienten Ressourcennutzung: Die resultierenden Spektren sind bei realwertigen Signalen symmetrisch, sodass die zweite Hälfte des Spektrums redundant ist. Dieses Verfahren wird in den zuvor erwähnten Arbeiten verwendet, geht jedoch mit einem unnötigen Verbrauch von Logikelementen und Chipfläche einher.

- **Packing-Verfahren für Real-valued-FFT:**
  Eine effizientere Methode zur Verarbeitung realer Eingangssignale ist das sogenannte Packing. Dabei werden reale Abtastwerte in gerade und ungerade Abtastwerte aufgeteilt: $x_e(n) = x(2n)$, $x_o(n) = x(2n + 1)$ und zu einem künstlich komplexen Wert zusammengeführt:
  $$z(n) = x_e(n) + j \cdot x_o(n) \tag{4.2}$$

Anschließend wird eine komplexe FFT auf $z(n)$ angewendet, was ein Spektrum $Z(k)$ ergibt. Dieses enthält jedoch nicht direkt das gesuchte Spektrum des ursprünglichen Signals $x(n)$. Um das echte Spektrum $X(k)$ zurückzugewinnen, ist eine Nachbearbeitung notwendig:

$$X(k) = \frac{1}{2}[Z(k) + Z^*(N-k) - j \cdot (Z(k) + Z^*(N-k)) \cdot W(k)] \tag{4.3}$$

Dabei ist:

- $Z(k)$: FFT des gepackten komplexen Signals
- $Z^*(N-k)$:konjungiert-komplexes Spiegelbild
- $W(k) = e^{-j2\pi k/N}$: Twiddle-Faktor

Mithilfe dieser Umrechnung wird das korrekte Spektrum X(k) des ursprünglichen realwertigen Signals berechnet. Da bei Realwerten aufgrund der Hermiteschen Symmetrie nur die ersten $N/2 + 1$ Frequenzkomponenten benötigt werden, lässt sich der Rechenaufwand nahezu halbieren. Dies macht die Methode besonders effizient für Hardware-Implementierungen.

Mel-Filter

Die optimierte Anwendung der Mel-Filterbank orientiert sich primär an der Arbeit von [49], in der die hardwareseitige Implementierung des MFCC-Algorithmus auf einem ASIC untersucht wurde. Ein zentrales Problem bei der Nutzung der Mel-Filterbank ist die hohe Rechenkomplexität, da jeder einzelne Mel-Filter auf das vollständige Frequenzspektrum angewendet werden muss. Dieser Zusammenhang wird in Gleichung 4.4 dargestellt:

$$\begin{bmatrix} S[0] \\ S[1] \\ \vdots \\ S[M-1] \end{bmatrix} = \begin{bmatrix} W_0(0) & W_0(1) & \dots & W_0(F/2-1) \\ W_0(0) & W_1(1) & \dots & W_1(F/2-1) \\ \vdots & \vdots & \vdots & \vdots \\ W_{M-1}(0) & W_{M-1}(1) & \dots & W_{M-1}(F/2-1) \end{bmatrix} \begin{bmatrix} X(0) \\ X(1) \\ \vdots \\ X(F/2-1) \end{bmatrix} \tag{4.4}$$

Dabei gilt:

- $W \in \mathbb{R}^{F \times M}$: Matrix der Mel-Filter
- $X \in \mathbb{R}^F$: Eingabedaten im Frequenzbereich
- $S \in \mathbb{R}^M$: resultierendes Mel-Spektrum

- M: Anzahl der Mel-Filter
- F: Anzahl der FFT-Punkte

Bei genauerer Analyse der einzelnen Filter einer Mel-Filterbank zeigt sich, dass der Großteil der Filterkoeffizienten den Wert Null annimmt und somit keinen Beitrag zur Berechnung des Mel-Spektrums leistet. Diese Nullwerte sind für die Signalverarbeitung irrelevant und führen in der klassischen Matrixdarstellung lediglich zu redundanten Speicher- und Rechenaufwänden.

Tabelle 4.3 zeigt exemplarisch, welche Frequenzbereiche bei einer Mel-Filterbank mit $M = 10$ Filtern und einem Spektrum mit $N = 128$ FFT-Bins tatsächlich Werte ungleich Null aufweisen. Deutlich wird: Jeder Filter ist nur in einem schmalen Frequenzbereich aktiv. Die Breite dieser aktiven Bereiche nimmt mit zunehmender Frequenz leicht zu, bleibt im Vergleich zur Gesamtanzahl der FFT-Punkte jedoch relativ gering.

Mel-Filter	non-zero Werte
1	$W_1(1) \sim W_1(4)$
2	$W_2(2) \sim W_2(6)$
3	$W_3(5) \sim W_3(10)$
4	$W_4(7) \sim W_4(14)$
5	$W_5(11) \sim W_5(19)$
6	$W_6(15) \sim W_6(25)$
7	$W_7(20) \sim W_7(32)$
8	$W_8(26) \sim W_8(40)$
9	$W_9(33) \sim W_9(51)$
10	$W_{10}(41) \sim W_{10}(63)$

Tabelle 4.3: Non-zero Werte der Mel-Filter 1 bis 10

Neben der zuvor beschriebenen Reduktion der Nullwerte in der Mel-Filterbank bestehen weitere Optimierungsmöglichkeiten, die sich aus einer strukturellen Analyse des Filteraufbaus ableiten lassen. Ein charakteristisches Merkmal der Mel-Filter ist ihre Überlappung: Die Filter sind so konstruiert, dass sie sich in ihren Übergangsbereichen teilweise überschneiden. Diese Eigenschaft kann gezielt genutzt werden, um Berechnungen effizienter zu gestalten.

Abbildung 4.7 veranschaulicht zwei benachbarte, überlappende Filter W_k und W_{k+1}. Der erste Filter beginnt bei L, erreicht sein Maximum bei $L + D$ und endet bei $L + 2D$. Der zweite Filter startet nahtlos bei $L+D$ erreicht seinen Höhepunkt bei $L+2D$ und endet bei

$L + 3D$. Ein spektraler Wert $X(j)$, der sich zwischen den Maxima beider Filter befindet, wird zur Berechnung beider Mel-Werte herangezogen.

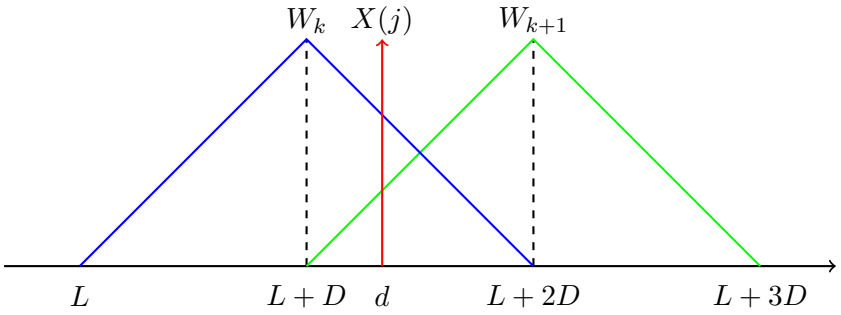

Abbildung 4.7: Berechnung der Mel-Werte

Die Auswertung von $X(j)$ für die beiden Filter erfolgt klassisch über zwei Multiplikationen mit linearen Gewichtungsfaktoren, wie in den Gleichungen 4.5 und 4.6 dargestellt:

$$E_k(j) = X(j)\frac{(L + 2D) - d}{(L + 2D) - (L + D)} \tag{4.5}$$

$$E_{k+1}(j) = X(j)\frac{d - (L + D)}{(L + 2D) - (L + D)} \tag{4.6}$$

Es zeigt sich jedoch, dass die Summe beider Beiträge stets dem Originalwert $X(j)$ entspricht:

$$E_k(j) + E_{k+1}(j) = X(j) \tag{4.7}$$

Daraus ergibt sich eine entscheidende Vereinfachung: Statt beide Multiplikationen auszuführen, genügt eine einzige Multiplikation (z.B. für $E_k(j)$) und eine anschließende Subtraktion zur Bestimmung von $E_{k+1}(j)$:

$$E_{k+1}(j) = X(j) - E_k(j) \tag{4.8}$$

Diese Erkenntnis reduziert die Anzahl der erforderlichen Multiplikationen drastisch. Während in der klassischen Umsetzung für jeden FFT-Punkt und jeden Mel-Filter eine Mul-

tiplikation notwendig ist (insgesamt $N \cdot M$), kann bei Anwendung dieser Optimierung die Anzahl der Multiplikationen auf maximal N reduziert werden.

Abbildung 4.8 zeigt die Umsetzung dieser optimierten Filterbank bei $M = 10$ Filtern. Grün markierte Filter sind direkt zu berechnen, rot markierte Filter ergeben sich durch Subtraktion gemäß Gleichung 4.8.

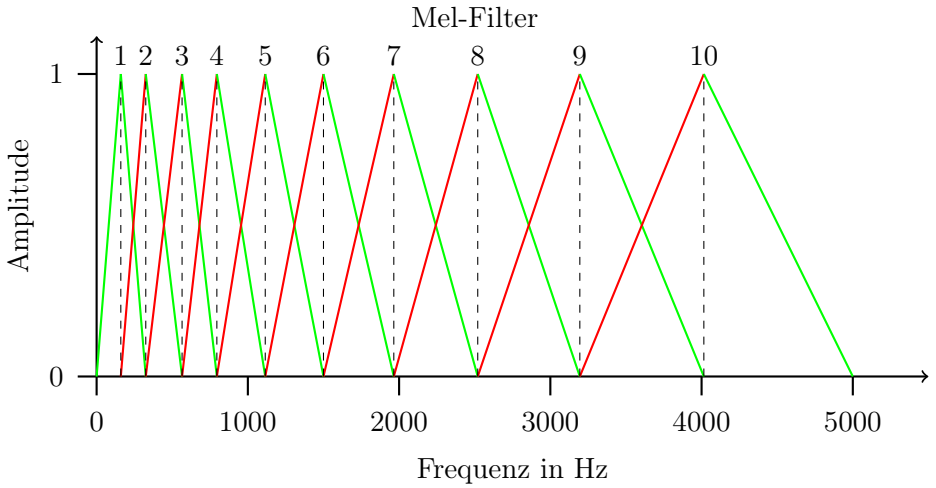

Abbildung 4.8: Diagram der Optimierten Filterbank

Diese Struktur erlaubt eine sequentielle Verarbeitung, bei der für jeden Spektrumswert $X(j)$ lediglich eine gewichtete Multiplikation sowie eine Subtraktion notwendig sind anstelle zweier getrennter Multiplikationen. Für Hardware-Implementierungen bedeutet dies eine erhebliche Einsparung an Rechenleistung und Stromverbrauch.

4.4.4 Audio Classification Unit

In der Audio Classification Unit erfolgt die Klassifizierung des Mel-Spektrums unter Zuhilfenahme eines Convolutional Neural Networks (CNN). Für die hardwarebasierte Realisierung eines solchen Netzwerks auf einem FPGA stehen verschiedene etablierte Toolchains und Frameworks zur Verfügung. Die Selektion des adäquaten Ansatzes wird maßgeblich von Faktoren wie Entwicklungsaufwand, Komplexität der Toolchain, Ressourcenverbrauch und verfügbarer Zeit determiniert. Im Folgenden werden vier gängige Ansätze gegenübergestellt: PYNQ, Vitis AI, FINN und hls4ml.

PYNQ

Mit PYNQ (Python Productivity for Zynq) können FPGAs über Python programmiert und gesteuert werden. Für CNNs gibt es mehrere einsatzbereite Overlays, die beispielsweise auf Xilinx-DPU-Paketen oder auf vereinfachten, selbst definierten Inferenzmodulen basieren. Die Entwicklung erfolgt primär in Python mit der Möglichkeit, vortrainierte Modelle (z. B. aus Keras oder PyTorch) einzubinden. PYNQ eignet sich insbesondere für das Prototyping, didaktische Szenarien und schnelle Evaluierungen, für die keine tiefgreifenden VHDL-/Verilog-Kenntnisse erforderlich sind.

Vorteile

- Schneller Einstieg, hohe Abstraktion
- Python-Umgebung mit Jupyter-Notebook-Unterstützung
- Gute Integration von vorkonfigurierten CNN-Overlays
- Sehr aktive Community und umfangreiche Beispiele

Nachteile

- Geringere Effizienz im Vergleich zu vollsynthetisierten Designs
- Begrenzter Einfluss auf Low-Level-Hardwaredetails

Vitis-AI

Vitis AI ist die offizielle Plattform von AMD/Xilinx für Deep Learning auf FPGAs und SoCs. Sie unterstützt optimierte Modelle (z. B. quantisierte TensorFlow-/Keras-Netze) und bietet einen dedizierten Compiler sowie eine Laufzeitumgebung für die DPU (Deep Learning Processing Unit). Vitis AI ist besonders für produktionsnahe, leistungsstarke Anwendungen geeignet.

Vorteile

- Hohe Performance bei unterstützten Netzen
- Professionelle Toolchain, kommerziell erprobt
- Effiziente Nutzung von Ressourcen durch spezialisierte Compiler

Nachteile

- Lange Einarbeitungszeit
- Eingeschränkte Modellkompatibilität
- Hoher Setup- und Konfigurationsaufwand

FINN
FINN ist ein Forschungsframework von Xilinx für binäre und quantisierte neuronale Netze (BNNs/QNNs) auf FPGAs. Es ermöglicht die vollständige End-to-End-Synthese quantisierter Netze bis hin zur Generierung von RTL-Modulen und IP-Cores. FINN ist insbesondere für ressourcenarme, latenzkritische Anwendungen geeignet.

Vorteile

- Sehr effiziente Umsetzung extrem kompakter Netzwerke
- Komplette Kontrolle über Quantisierung und Hardware

Nachteile

- Fokus auf stark quantisierte Netze (z.B. INT2, INT1)
- Begrenzter Support für konventionelle CNN-Architekturen
- Komplexe Toolchain, stark forschungsnah

hls4ml
hls4ml ist ein Open-Source-Framework zur Übersetzung einfacher neuronaler Netze in High-Level-Synthese-Code (C++), der auf FPGAs lauffähig ist. Es wurde ursprünglich für Anwendungen in der Hochenergiephysik entwickelt, inzwischen unterstützt es aber auch kleine CNNs und MLPs.

Vorteile

- Generiert HLS-Code aus Keras/PyTorch-Modellen
- Gute Dokumentation und niedriger Einstieg

Nachteile

- Nur einfache Netzwerke unterstützt (keine komplexen CNNs)
- Eingeschränkte Hardwareoptimierungsmöglichkeiten
- Kein direktes Deployment-Tool wie bei PYNQ oder Vitis AI

Auswahl
Aufgrund des begrenzten Zeitrahmens und des im Fokus stehenden Prototypings wurde die Umsetzung des CNN mit PYNQ realisiert. Die einfache Integration bestehender Modelle, die verfügbare Python-Infrastruktur und die zahlreichen vorhandenen Beispiele ermöglichten eine zügige Entwicklung ohne tiefen Einstieg in komplexe Toolchains wie Vitis AI oder FINN.

Zudem bietet PYNQ ausreichende Flexibilität für Evaluierungs- und Vergleichszwecke, bei gleichzeitig minimalem Setup-Aufwand. Im Vergleich zu hls4ml oder FINN entfällt die Notwendigkeit, Modelle zu quantisieren oder vollständig neu zu strukturieren. Damit stellt PYNQ einen pragmatischen Mittelweg zwischen Entwicklungsaufwand und Funktionalität dar und ist somit am besten für die Ziele dieser Arbeit geeignet.

4.5 Vollständige Datenverarbeitung

Auf Basis der zuvor entwickelten Teilkonzepte zum modularen Aufbau der Datenverarbeitung kann nun eine vollständige Datenverarbeitung für das angestrebte SoC-System abgeleitet werden. Die Umsetzung orientiert sich dabei sowohl an den funktionalen Anforderungen aus Kapitel 3.3 als auch an den in den vorangegangenen Abschnitten dargelegten strukturellen und technischen Rahmenbedingungen.

- AIU Audio Input Unit
- SFU Signal Framing Unit
- MPU Mel Processing Unit
- ACU Audio Classification Unit

Für drei der vier Module wurden jeweils zwei alternative Umsetzungskonzepte entwickelt, um die Vor- und Nachteile unterschiedlicher Entwicklungsmethoden systematisch vergleichen zu können. Die ACU bildet hierbei eine Ausnahme, da sie ausschließlich über die PYNQ-Plattform realisiert wird. Aus der Kombination der möglichen Implementierungen ergeben sich insgesamt acht Varianten, die in Tabelle 4.4 dargestellt sind.

Variante	AIU	SFU	MPU	ACU
1	VHDL	HLS	HLS 1	PYNQ
2	HLS	HLS	HLS 1	PYNQ
3	VHDL	HLS	HLS 2	PYNQ
4	HLS	HLS	HLS 2	PYNQ
5	VHDL	RPU	HLS 1	PYNQ
6	HLS	RPU	HLS 1	PYNQ
7	VHDL	RPU	HLS 2	PYNQ
8	HLS	RPU	HLS 2	PYNQ

Tabelle 4.4: Aufschlüsselung der Varianten

Es ist zu beachten, dass im finalen Gesamtsystem auf Basis der PYNQ-Plattform keine RPU eingesetzt werden kann. Eine ursprünglich als Alternative vorgesehene Umsetzung der SFU auf der RPU ist daher nicht realisierbar. Folglich werden im Folgenden ausschließlich die Varianten 1 bis 4 betrachtet.

5 Implementierung

Das folgende Kapitel behandelt die Implementierung des zuvor entwickelten Konzepts. Zunächst werden die einzelnen Komponenten realisiert, bevor diese in einem zweiten Schritt zu einer vollständigen Datenverarbeitung zusammengeführt werden.

5.1 Hardware

Für die Implementierung des Hardwaresystems wird Wie in Kapitel 4.2.5 beschrieben, das Kria KV260 als zentrales System-on-Chip eingesetzt. Das Board ist die Basisplattform, auf der die Datenverarbeitung realisiert wird. Der Audioeingang wird über das I2S2-Pmod-Modul von Digilent angeschlossen, das direkt in den Pmod-Anschluss des KV260 gesteckt wird. Die empfangenen Audiodaten werden über die Schnittstelle in das SoC eingespeist.

Abbildung 5.1: Implementiertes Hardwaresystem

Die Ausgabe der Ergebnisse erfolgt über ein oberhalb des Boards montiertes Display, das per HDMI angesteuert wird. Das Display ist direkt mit dem HDMI-Ausgang des KV260 verbunden und zeigt die Mel-Spektren und Klassifikationsergebnisse an.

Alle Komponenten sind in einem Gehäuse integriert, das die physische Stabilität des Systems gewährleistet und gleichzeitig den kompakten Aufbau unterstützt. Das Gehäuse wurde in Fusion360 konstruiert und anschließend mittels 3D-Druck gefertigt, sodass eine präzise Passform und eine flexible Montage aller Bauteile gewährleistet ist. Die vollständige Hardware, wie es für die Implementierung der Datenverarbeitung zusammengesetzt wurde, ist in Abbildung 5.1 dargestellt

5.2 Audio Input Unit

Die Implementierung der Audio Input Unit (AIU) erfolgt, wie in Kapitel 4.4.1 beschrieben, sowohl in VHDL als auch in HLS.

Die Realisierung des I2S-Protokolls basiert auf dem ersten Labor für Digitale Systeme von Prof. Leutelt, in dem ein 16-Bit-Audiocodec unter Verwendung des I2S-Protokolls in Betrieb genommen wurde. Das in dieser Arbeit verwendete I2S2-Pmod arbeitet jedoch mit einer Auflösung von 24 Bit, während im Labor lediglich 16 Bit verarbeitet wurden. Daher wurden die Taktraten sowie die Einlesefunktion an einen 32-Bit-Frame mit 24 Nutzbits angepasst. Das ursprüngliche Labormodul führte die Audioein- und -ausgabe parallel aus. Für die Implementierung der AIU wurde anstelle einer direkten Datenausgabe ein AXI-Stream-Interface integriert, das mithilfe des von Vivado bereitgestellten IP-Wizards realisiert wurde.

Die Implementierung in HLS basiert auf denselben konzeptionellen Grundlagen, da sich die Funktionsweise des I2S-Protokolls unabhängig von der verwendeten Beschreibungssprache nicht verändert. Lediglich die Anbindung an das AXI-Stream-Interface unterscheidet sich, da in HLS lediglich spezifiziert werden muss, dass die Daten als Stream ausgegeben werden sollen.

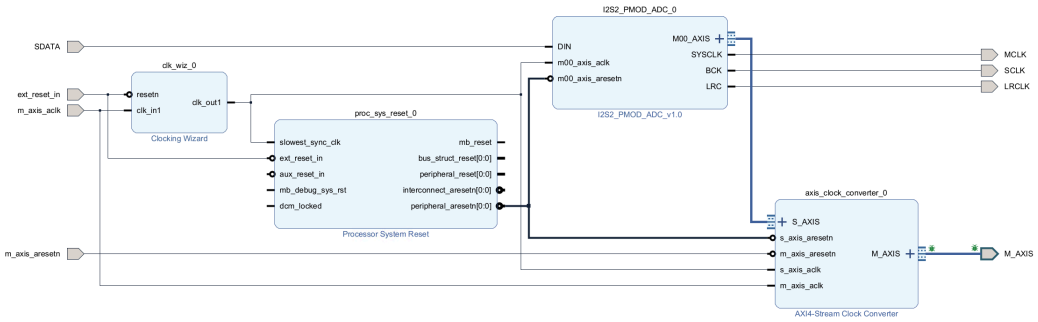

Abbildung 5.2: Blockschaltbild AIU

Da weder das VHDL- noch das HLS-Design aus dem internen Standardtakt von 100 MHz direkt eine Abtastrate von 44,1 kHz erzeugen können, wird der Takt mithilfe des Clocking Wizard gemäß Gleichung 5.1 auf 90,3,MHz reduziert. Die Datenübertragung über das AXI-Stream-Interface erfolgt unter Verwendung eines AXI-Stream-CDC-Moduls, das einen Übergang zwischen unterschiedlichen Taktdomänen ermöglicht. Das resultierende Blockschaltbild ist in Abbildung 5.2 dargestellt.

$$\text{Grundtakt} = \text{Zieltakt} \cdot \text{Taktteiler} = 44,1\text{kHz} \cdot 2048 = 90,3\text{MHz} \tag{5.1}$$

5.3 Signal Framing Unit

In Abschnitt 4.4.2 wurde erläutert, dass die Signal Framing Unit (SFU) die eingehenden Daten zunächst filtert, anschließend unterabgetastet, zu Frames zusammenfasst und schließlich ein Hanning-Fenster auf diese anwendet. Der entsprechende Ablauf ist in Abbildung 5.3 dargestellt.

Die Filterung und Unterabtastung erfolgen mithilfe eines Multiratenfilters, dessen Berechnungen abwechselnd bei jedem Aufruf durchgeführt werden. Das Filterergebnis wird anschließend in ein Schieberegister geschrieben. Aus diesem Register wird der auszugebende Wert entnommen und mit dem entsprechenden Koeffizienten des Hanning-Fensters multipliziert. Die Fensterkoeffizienten werden dabei abwechselnd als Real- oder Imaginärwerte zwischengespeichert. Sobald ein vollständiger komplexer Wert berechnet ist, wird dieser gespeichert und nach Abschluss jedes Frames als Datenblock ausgegeben. In

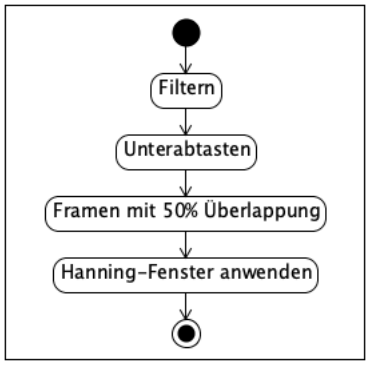

Abbildung 5.3: Ablauf AIU

Abbildung 5.4 ist dieser Ablauf in Form eines Aktivitätsdiagramms dargestellt. Die Implementierung sowohl auf der RPU als auch in HLS basiert auf diesem Konzept. In den nachfolgenden Abschnitten werden die wesentlichen Unterschiede zwischen den beiden Implementierungsansätzen detailliert erörtert.

5.3.1 RPU

Für die Implementierung der SFU auf der RPU wird ein DMA eingesetzt, um Audiodaten von der AIU zu empfangen. Sobald neue Daten vorliegen, löst der DMA einen Interrupt aus, woraufhin der Prozessor das entsprechende Programm abarbeitet.

Die für die Berechnungen benötigten Koeffizienten des FIR-Filters sowie die Werte des Hanning-Fensters wurden mithilfe eines MATLAB-Skripts generiert und in einer Header-Datei hinterlegt. Die verarbeiteten Daten werden im DDR4-RAM abgelegt und anschließend wahlweise über einen weiteren DMA ausgegeben oder per Inter-Processor-Interrupt (IPI) an einen anderen Prozessor weitergeleitet, abhängig vom jeweiligen Verwendungszweck.

Die Festlegung der Koeffizientenanzahl auf 44 erfolgte auf Grundlage der im Konzept erstellten Tabelle 4.2 und wurde unter Berücksichtigung von Laufzeittests auf dem Prozessor validiert. Die gewählten Parameter entsprechen einer Dämpfung von 40 dB, einer Durchlassfrequenz $f_{pass} = 9kHz$ sowie einer Sperrfrequenz $f_{stop} = 11,025kHz$. Damit wird ein Kompromiss zwischen ausreichender Filtergüte und effizienter Prozessorlaufzeit realisiert.

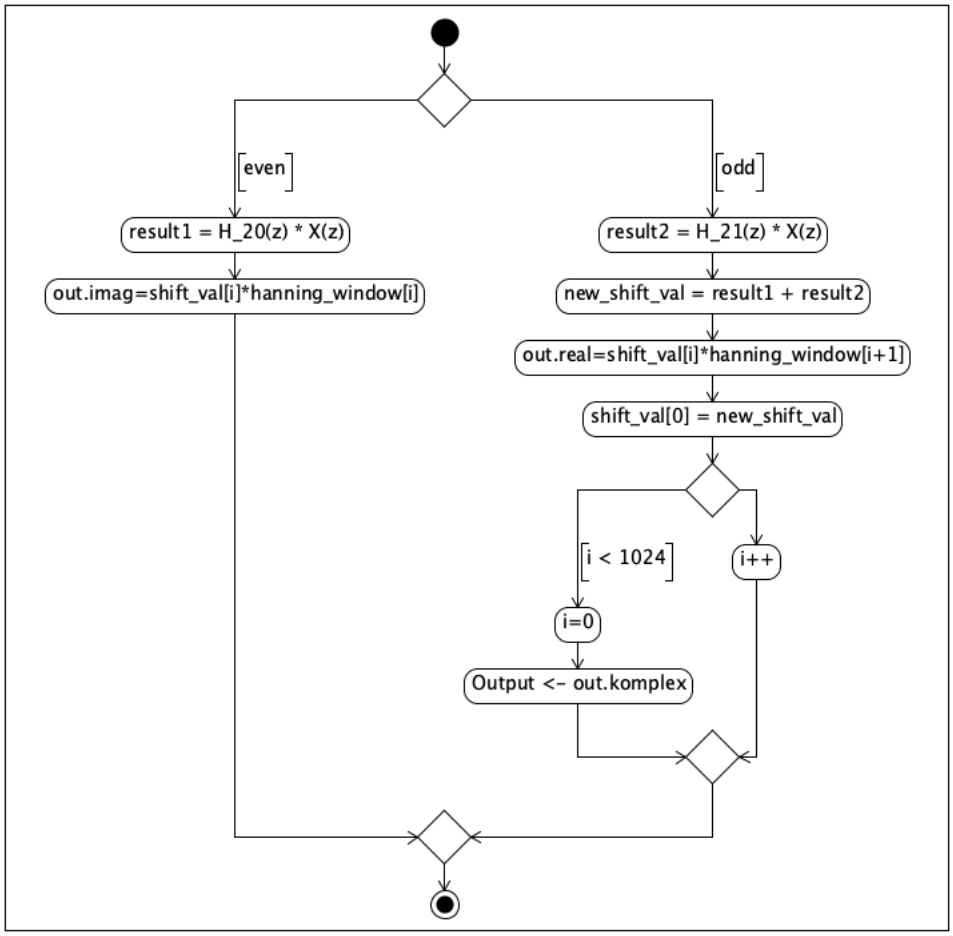

Abbildung 5.4: Aktivitätsdiagramm AIU

5.3.2 HLS

Im Rahmen der HLS-Implementierung werden die relevanten Eingangsdaten über ein AXI-Stream-Interface bereitgestellt. Die Komponente ist so konfiguriert, dass sie ausschließlich bei Vorliegen neuer Daten aktiv wird. Dies wird durch den Betrieb des AXI-Stream-Inputs im Blocking-Modus erreicht, wodurch die Ausführung des Blocks unmittelbar an den Empfang neuer Daten gekoppelt ist. Zusätzliche Steuersignale sind daher nicht erforderlich. Die Entfernung dieser Signale wird mithilfe des in Listing 5.1 dargestellten Pragmas realisiert.

Listing 5.1: Pragma zur Entfernung der Steuersignale

```
1  #pragma HLS INTERFACE mode=ap_ctrl_none port = return
```

Wie bei der RPU-Implementierung werden die für die Filterung erforderlichen FIR-Filterkoeffizienten in einer mithilfe eines MATLAB-Skripts generierten Header-Datei hinterlegt. Die Werte des Hanning-Fensters werden hingegen erst während der Synthese erzeugt, wodurch der Bedarf an externen Hilfsmitteln reduziert wird. Dieses Verfahren ist in Listing 5.2 dargestellt. Alternativ besteht jedoch auch die Möglichkeit, für das Hanning-Fenster eine separate Header-Datei zu verwenden.

Listing 5.2: ROM generierung für das Hanning-Fenster

```
1  ap_ufixed<16, 1> hanning_window[N_FFT];
2  for (int i = 0; i < N_FFT; i++) {
3      hanning_window[i] = 0.5*(1-std::cos((2*PI*i)/(N_FFT-1)));}
```

Für die weitere Verarbeitungskette wird der Code der RPU übernommen. Die Ausgabe unterscheidet sich jedoch dahingehend, dass die Daten direkt über ein AXI-Stream-Interface übertragen werden. Da die Werte potenziell von einem DMA-Modul empfangen werden sollen, muss der letzte Wert eines Blocks mit einem TLAST-Signal versehen werden, um das Ende des Datenblocks zu kennzeichnen.

5.4 Mel Processing Unit

Die Mel Processing Unit (MPU) wird, wie in Abschnitt 4.4.3 beschrieben, sowohl in einer Standardvariante als auch in einer optimierten Version mittels HLS implementiert. Die Untergliederung der Datenverarbeitung folgt dabei den bereits im Konzept festgelegten Untermodule, um eine strukturierte und effiziente Bearbeitung der Daten zu gewährleisten. Da die ein- und ausgehenden Daten in Form von Blöcken übertragen werden, sind ein *Inputdatamover* für den Empfang und ein *Outputdatamover* für die Übertragung erforderlich. Diese Unterteilung ist in Abbildung 5.5 veranschaulicht. Im Folgenden werden die Implementierungen der einzelnen Untermodule im Detail beschrieben.

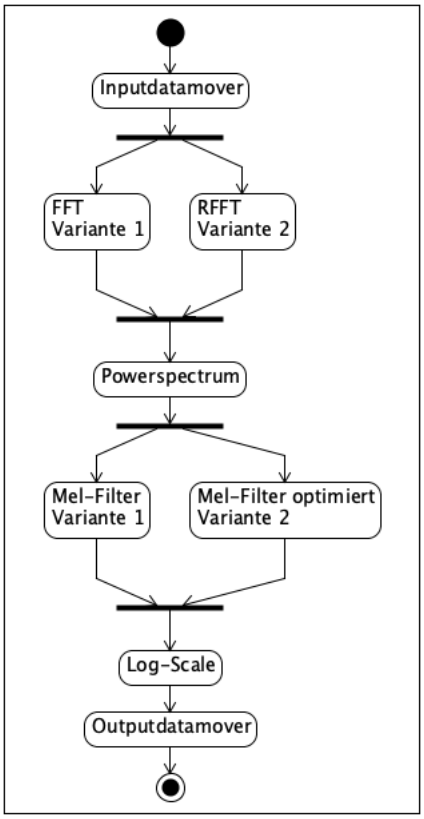

Abbildung 5.5: Aktivitätsdiagramm der MPU

5.4.1 Inputdatamover

Der Input-Datamover ist für das Einlesen der Daten über das AXI-Stream-Interface und deren Bereitstellung zur weiteren Verarbeitung verantwortlich. Hierbei werden die seriellen Stream-Daten in einer Schleife parallel in einem Array zwischengespeichert. Der beschriebene Ablauf ist in Form eines Aktivitätsdiagramms in Abbildung 5.6 dargestellt.

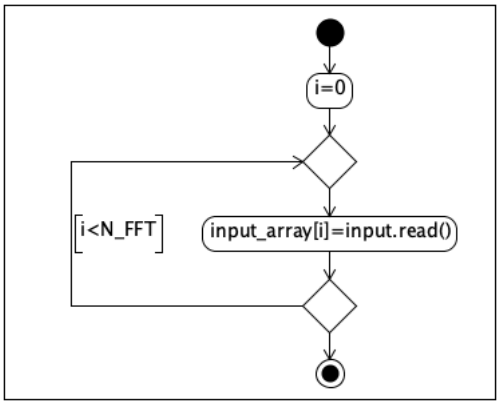

Abbildung 5.6: Aktivitätsdiagramm Inputdatamover

5.4.2 FFT

Für die Implementierung der FFT unter HLS wird ein vorhandener IP-Core [12] verwendet, der direkt aus dem HLS-Code heraus aufgerufen werden kann [14]. Hierzu wird zunächst ein spezieller HLS-FFT-Header eingebunden. Der Aufruf der FFT erfolgt anschließend gemäß dem in Listing 5.3 dargestellten Beispiel.

Listing 5.3: FFT-IP-Core in HLS

```
1 #include "hls_fft.h"
2 hls::fft<config1>(xn, xk, &fft_status, &fft_config);
```

Die Größe der FFT ergibt sich automatisch aus der Länge des Eingangsarrays *xn*. Abhängig von der Größe kann es erforderlich sein, zusätzliche HLS-Pragmas zu verwenden, um die Ressourcenzuordnung der FFT explizit festzulegen. Ohne eine solche Angabe kann es in Vitis zu undefiniertem Verhalten oder Laufzeitfehlern kommen.

Über den Parameter *fft_config* lassen sich wesentliche Eigenschaften der FFT-Instanz zur Laufzeit konfigurieren, beispielsweise die Transformationsrichtung (Vorwärts/Rückwärts) sowie das Rundungsverhalten innerhalb des IP-Cores. Der Rückgabewert *fft_status* liefert unter anderem Informationen über mögliche Überläufe während der Berechnung und ermöglicht somit die Überwachung der numerischen Stabilität. Weitere Details zur Konfiguration und zum Verhalten des FFT-IP-Cores sind im entsprechenden Benutzerhandbuch [12] dokumentiert.

5.4.3 Real-Valued FFT

Für die optimierte Variante wird, wie in Kapitel 4.4.3 beschrieben, eine Real-Valued FFT verwendet. Dieses Verfahren basiert auf einer zusätzlichen Nachbearbeitungsstufe, die an eine konventionelle komplexe FFT angehängt wird. Hierfür wird zunächst der zuvor beschriebene FFT-IP-Core eingesetzt, während die Nachverarbeitung anschließend mittels HLS-Code realisiert wird. Der dazu notwendige Ablauf ist in Form eines Aktivitätsdiagramms in Abbildung 5.7 dargestellt.

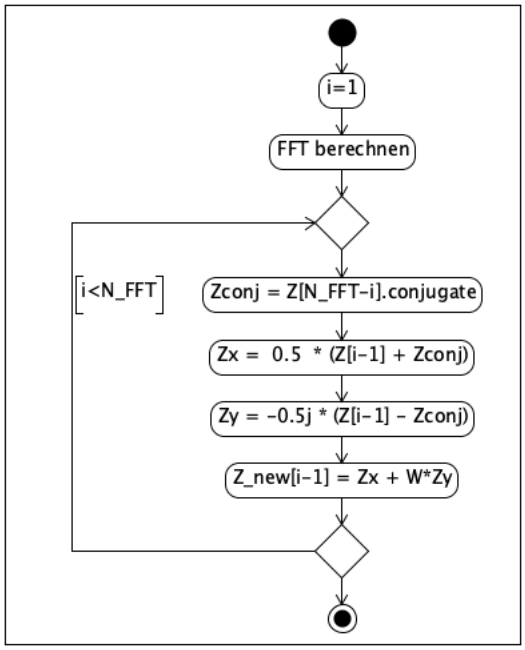

Abbildung 5.7: Aktivitätsdiagramm RFFT

Die für die FFT erforderlichen Twiddle-Faktoren werden, wie in Abschnitt 5.3 erläutert, bereits zur Synthesezeit generiert. Da es sich hierbei um komplexe Werte handelt, müssen Real- und Imaginärteil separat gespeichert und erst bei Bedarf zu komplexen Werten zusammengefügt werden, wie in Listing 5.4 dargestellt. Andernfalls kann es während der Synthese zu einer fehlerhaften Interpretation kommen, bei der die Werte in jedem Durchlauf neu berechnet werden.

Listing 5.4: Twiddle-Faktor in HLS

```
1 data_in_t twiddle_real[FFT_LENGTH];
2 data_in_t twiddle_imag[FFT_LENGTH];
3 for (int i = 0; i < (FFT_LENGTH); i++) {
4  twiddle_real[i] = std::cos((2*PI*i)/(FFT_LENGTH));
5  twiddle_imag[i] = std::sin((2*PI*i)/(FFT_LENGTH));
6  }
```

5.4.4 Leistungsspektrum

Wie in Kapitel 2.3 beschrieben, werden aus den komplexen Ausgangswerten der FFT bzw. RFFT die entsprechenden Leistungsspektren berechnet. Hierfür werden jeweils zwei Multiplikationen und eine Addition durchgeführt. Um sicherzustellen, dass auch sehr kleine Werte nicht durch Rundungsfehler verloren gehen, werden die 16-Bit-Komplexwerte in 32-Bit-Realwerte umgewandelt. Anschließend werden diese Realwerte als vorzeichenlose Ganzzahlen (Unsigned) gespeichert, um den vollständigen Dynamikbereich auszuschöpfen.

5.4.5 Mel-Filter

Die zentrale Komponente der Mel Processing Unit (MPU) ist die Mel-Filtereinheit, in der die namensgebenden Mel-Filter auf das berechnete Spektrum angewendet werden. Die Implementierung erfolgt, wie in Kapitel 4.4.3 beschrieben, in zwei Varianten: die erste in klassischer Form und die zweite in optimierter Ausführung.

Variante 1
Bei der klassischen Variante werden die Mel-Filter durch $N \cdot M$ Multiplikationen berechnet, das heißt, jeder Filter wird auf das gesamte Spektrum angewendet. Der Ablauf dieser Berechnung ist in Form eines Aktivitätsdiagramms in Abbildung 5.8 dargestellt.

Die verwendete Filterbank wurde mithilfe eines MATLAB-Skripts generiert. Dabei wurden die erforderlichen Normalisierungsfaktoren direkt in die Filterkoeffizienten integriert, sodass keine zusätzliche Skalierung erforderlich ist [**?**].

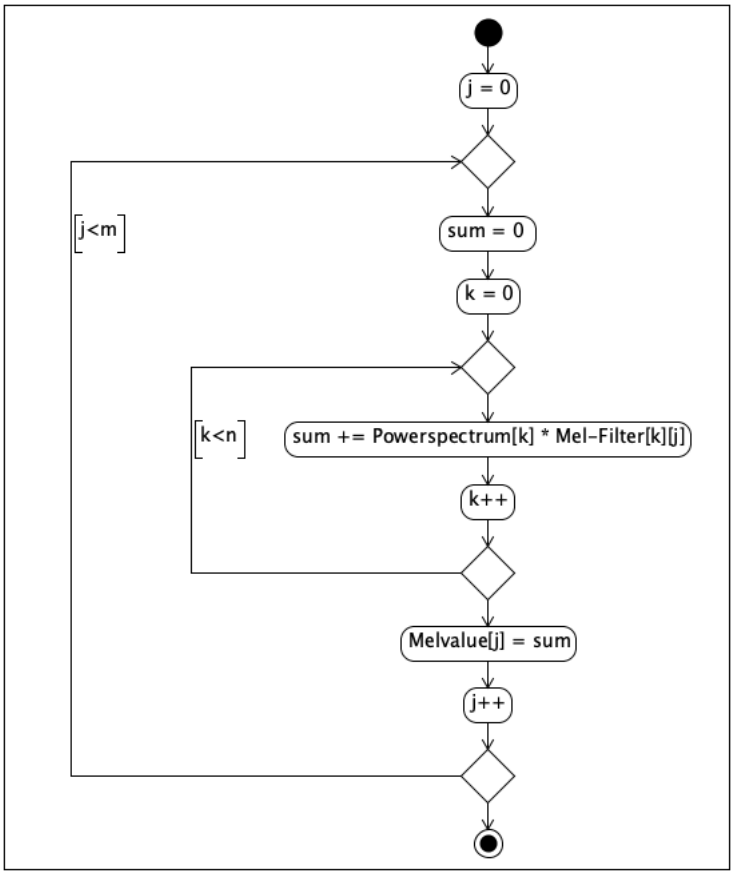

Abbildung 5.8: Aktivitätsdiagramm Mel-Filter 1

Variante 2

Bei der zweiten Variante kommt die in Abschnitt 4.4.3 vorgestellte optimierte Filterbank zum Einsatz. Durch die Wiederverwendung überlappender Filteranteile reduziert sich die Anzahl der erforderlichen Multiplikationen auf lediglich N operationen, da nur ein Filterwert pro spektralem Punkt direkt berechnet wird. Der Ablauf dieses Verfahrens ist in Form eines Aktivitätsdiagramms in Abbildung 5.9 dargestellt.

Da die benachbarten Filteranteile anteilig wiederverwendet werden, müssen zusätzliche Skalierungsfaktoren berücksichtigt werden, um das korrekte Ergebnis zu rekonstruieren. Insgesamt ergibt sich somit ein Rechenaufwand von $N + M$ Multiplikationen. Sowohl die benötigte Filterbank als auch die Skalierungsfaktoren werden, wie zuvor, mithilfe eines entsprechenden MATLAB-Skripts [30] generiert und in einer Header-Datei hinterlegt.

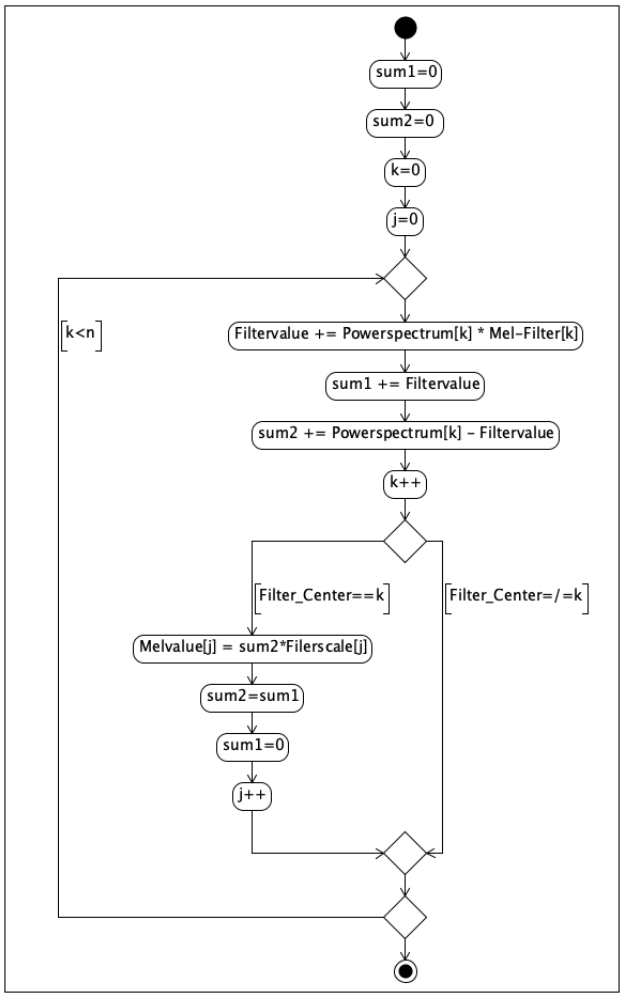

Abbildung 5.9: Aktivitätsdiagramm Mel-Filter 2

5.4.6 Logarithmisches Amplitudenspektrum

Nach der Berechnung des Mel-Spektrums wird aus diesem das logarithmische Amplitudenspektrum abgeleitet. Hierbei wird auf jeden Wert des Mel-Spektrums eine logarithmische Funktion angewendet. Für die Berechnung kommt die HLS-Math-Bibliothek zum Einsatz, die die gängigen Standard-Mathematikfunktionen hardwareoptimiert bereitstellt.

5.4.7 Outputdatamover

Der Outputdatamover ist für das Zusammenführen des aktuellen Frames mit dem restlichen Spektrum sowie für die anschließende Ausgabe verantwortlich. Hierzu werden die Werte zunächst in ein Schieberegister eingelesen und anschließend in einer Schleife über das AXI-Stream-Interface ausgegeben. Die Reihenfolge der Ausgabe entspricht dem AXI-Stream-Videoprotokoll, das heißt von oben links bis unten rechts, jeweils Zeile für Zeile. Zu Beginn einer Übertragung wird das Signal *T_USER* auf High gesetzt, während am Ende jeder Zeile ein *T_LAST*-Signal generiert wird. Der Ablauf dieses Prozesses ist in Abbildung 5.10 als Aktivitätsdiagramm dargestellt.

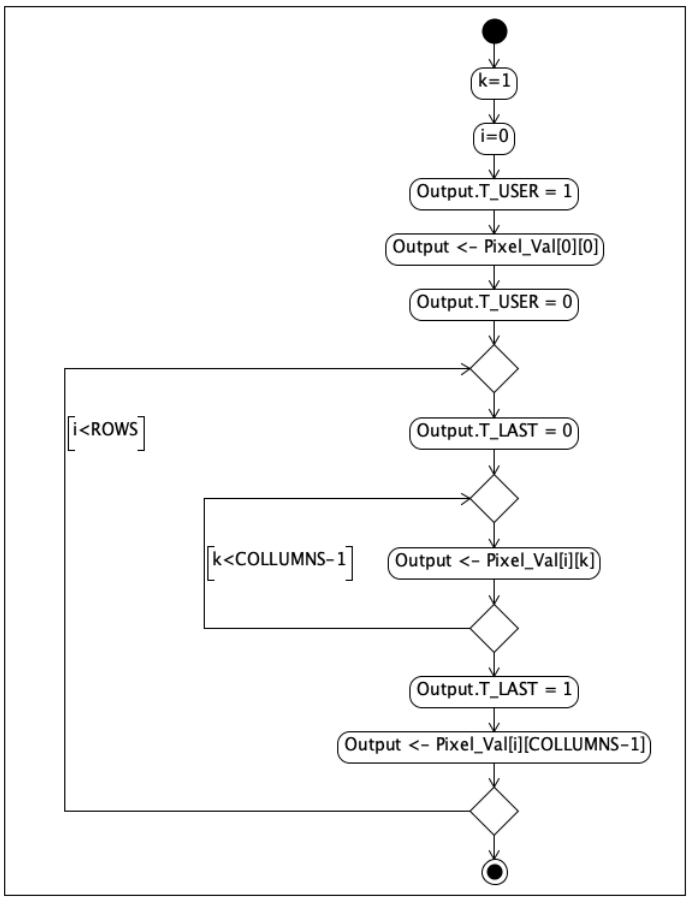

Abbildung 5.10: Aktivitätsdiagramm Outputdatamover

5.5 Audio Classification Unit

Die Implementierung der ACU erfolgt gemäß dem in Kapitel 4.4.4 beschriebenen Konzept unter Verwendung der PYNQ-Plattform. Zu diesem Zweck wurde ein Jupyter-Notebook erstellt, in dem zunächst das vortrainierte neuronale Netz geladen wird. Anschließend wird eine Funktion implementiert, die das berechnete Spektrum an das CNN übergibt und als Ausgabe einen Ergebnisvektor mit den zehn ermittelten Klassifikationswahrscheinlichkeiten zurückliefert. Der beschriebene Ablauf ist im Aktivitätsdiagramm in Abbildung 5.11 dargestellt.

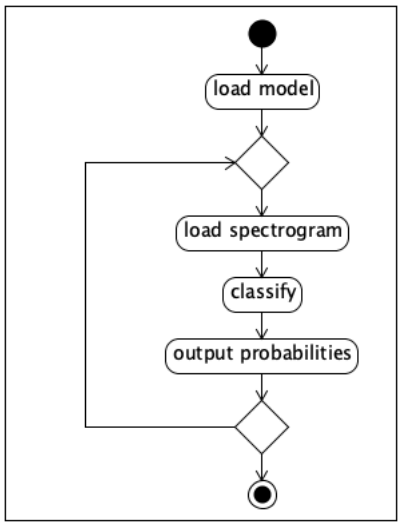

Abbildung 5.11: Aktivitätsdiagramm ACU

5.6 Vollständige Datenverarbeitung

Die einzelnen Module — AIU, SFU, MPU und ACU — wurden zunächst separat entwickelt und validiert. Wie in Kapitel 4.5 erläutert, werden diese Module anschließend zu vier unterschiedlichen Systemvarianten zusammengeführt. Die vollständige Implementierung gliedert sich in ein Hardware- und ein Softwaredesign. Das Hardwaredesign beschreibt die physische Architektur des SoC, während die Softwarearchitektur die auf der PYNQ-Plattform erforderlichen Rahmenbedingungen und Steuerungsmechanismen bereitstellt.

5.6.1 Hardwaredesign

Die Varianten unterscheiden sich durch die Kombination der verschiedenen Modulvarianten, wie in Tabelle 4.4 dargestellt. Da alle Module als austauschbare Blöcke implementiert wurden, lässt sich die vollständige Datenverarbeitung in Form eines allgemeinen Blockschaltbilds abbilden. Abbildung 5.12 zeigt beispielhaft die Variante 1. Die weiteren Varianten wurden analog implementiert. Abschließend wurden aus den Designs die Bitstream-Datei (.bit) sowie die Hardwarebeschreibung (.hwh) generiert, um sie später in PYNQ zu verwenden.

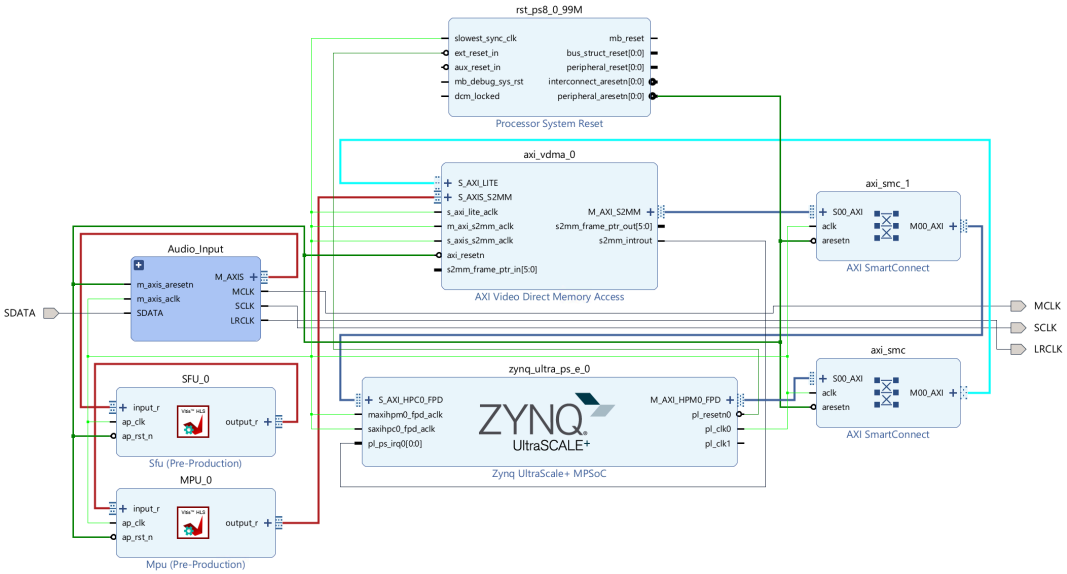

Abbildung 5.12: Blockdiagramm der vollständigen Datenverarbeitung

5.6.2 Softwaredesign

Die Software wird mithilfe von PYNQ in einem Jupyter Notebook ausgeführt. Nach dem Laden der Hardwarekonfiguration wird zunächst die ACU initialisiert. Anschließend erfolgt die Eingabe über eine Video-DMA-Schnittstelle, welche die von der MPU berechneten Mel-Spektren direkt in den Speicherbereich der ACU überträgt. Die Datenübertragung ist durch Interrupts synchronisiert, sodass die ACU unmittelbar nach dem Eintreffen neuer Daten automatisch mit der Verarbeitung beginnt. Die Klassifikationsergebnisse werden anschließend auf dem angeschlossenen Display ausgegeben. Dabei werden das aktuelle Mel-Spektrum, ein Bild des erkannten Vogels sowie die zugehörigen Wahrscheinlichkeiten dargestellt. Sowohl die Konfiguration der DMA-Schnittstelle als auch die Implementierung der Interrupt-Handler sind integrale Bestandteile des Designs und gewährleisten eine zuverlässige sowie latenzarme Datenverarbeitung. Der Gesamtablauf ist in Abbildung 5.13 dargestellt.

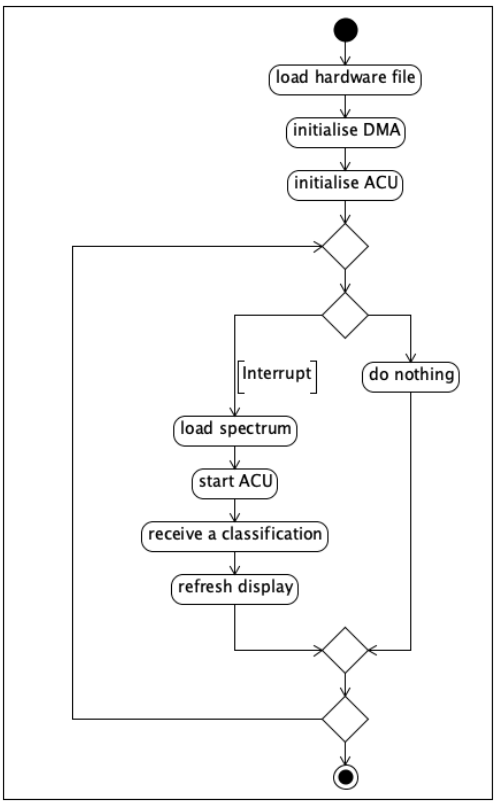

Abbildung 5.13: Aktivitätsdiagramm der Software

6 Ergebnisse und Analyse

In diesem Kapitel werden die Ergebnisse der in dieser Arbeit entwickelten und untersuchten Konzepte präsentiert und bewertet. Zunächst erfolgt eine separate Analyse der einzelnen Module sowie ihrer jeweiligen Implementierungsvarianten. Anschließend werden die verschiedenen Varianten der vollständigen Datenverarbeitung gegenübergestellt, bevor eine abschließende Bewertung vorgenommen wird. Ziel dieser Untersuchung ist es, den Beitrag jedes Moduls zur Performance sowie zum Ressourcenverbrauch zu quantifizieren und diese Ergebnisse im Hinblick auf die in Kapitel 3.3 definierten Anforderungen einzuordnen.

6.1 Audio Input Unit

Die AIU wurde in zwei Varianten realisiert: einmal als klassische VHDL-Implementierung und einmal mit HLS. Das Ziel bestand darin, zu untersuchen, inwieweit sich HLS als Alternative zu VHDL-Architekturen, insbesondere bei einfachen, protokollnahen Modulen, eignet.

6.1.1 Funktionale Validierung

Die Audio Input Unit liest das Audiosignal über das PMOD-I2S2-Interface ein und stellt die Daten als 16-Bit-AXI-Stream bereit. Damit wird die Anforderung H-5 erfüllt. Sowohl die klassische VHDL-Implementierung als auch die auf HLS basierende Variante lieferten stabile und fehlerfreie Ergebnisse. Die Datenausgabe erfolgte in beiden Fällen mit einer Verzögerung von einem Takt. Es konnten keine Signalverluste oder Synchronisationsprobleme festgestellt werden.

Die funktionale Validierung erfolgte in mehreren Stufen. Zunächst wurde die Latenz in einer Vivado-Testbench simuliert und verifiziert. Anschließend erfolgte die Überprüfung

auf der realen Hardware mithilfe eines integrierten Logic Analyzers (ILA), mit dem die internen Signalflüsse aufgezeichnet und analysiert wurden. Ergänzend wurden die vom I2S2-Modul erzeugten Clock-Signale mit einem Oszilloskop gemessen, um deren Frequenz und Stabilität zu kontrollieren.

6.1.2 Messergebnisse und Ressourcenauslastung

Die Ressourcenauslastung wurde nicht den HLS-Schätzungen entnommen, da diese laut User Guide [14] nur eingeschränkt zuverlässig sind. Stattdessen erfolgte die Bestimmung auf Basis der vollständigen Implementierung in Vivado, womit realistische Werte für das Gesamtdesign ermittelt werden konnten.

Die Ergebnisse sind in Tabelle 6.1 dargestellt. Dabei zeigt sich, dass die HLS-Variante etwas mehr LUTs beansprucht, während die VHDL-Variante geringfügig mehr Flip-Flops benötigt.

Tabelle 6.1: Vergleich des Ressourcenverbrauchs zwischen HLS und VHDL

Resourcen	DSP Slices	LUT	FF	BRAM
VHDL	0	12	77	0
HLS	0	36	75	0

6.1.3 Analyse

Beide Implementierungen (VHDL und HLS) realisieren die AIU funktional identisch und mit einer Latenz von einem Takt. Ein Zeitvorteil durch HLS ergab sich nicht. Die Ursachen hierfür lassen sich wie folgt einordnen:

Aufgabencharakter: protokollnahe Low-Level-Logik
Die AIU besteht im Wesentlichen aus zustandsbasierten Sequenzen zur Bit-/Word-Ausrichtung (I2S) und einer eng getakteten Handshake-Anbindung (AXI-Stream). Solche endlichen Automaten mit bitgenauer Taktbeziehung profitieren kaum von den Abstraktionen der HLS-Planung und -Synthetisierung. In HLS mussten Zustandsübergänge, Takt-Enable, Reset-Semantik und Handshakes ebenso explizit modelliert werden wie in VHDL – HLS fungierte hier faktisch nur als alternative Beschreibungssprache.

Latenz ist I/O-gebunden, nicht rechengebunden
Die dominierenden Zeitkonstanten entstehen durch das serielle I2S-Einlesen und den AXI-Stream-Takt. Da keine tiefen Rechenpipelines vorliegen, kann HLS keine nennenswerte Parallelisierung heben. Entsprechend bleibt die Latenz in beiden Varianten auf demselben Niveau.

Verifikation und Entwicklungsaufwand
Zyklusgenaue Testbenches lassen sich in VHDL sehr direkt umsetzen. Stimuli und Checker sind eins-zu-eins auf die Signalflanken ausgerichtet. In HLS erfordert die Brücke zwischen C-/C++-Testbench, Co-Simulation und RTL zusätzliche Infrastruktur, ohne die Verifikation zu vereinfachen.

Tool-Flow und Umgebung Elemente wie Clocking-Wizard, AXI-Stream-CDC und Pin-Zuordnung bleiben unabhängig von HLS erforderlich und bestimmen einen Großteil des Integrationsaufwands. Dadurch nivellieren sich potenzielle Produktivitätsgewinne der HLS-Variante in diesem Kontext.

Fazit
Für protokollnahe, timingkritische Schnittstellen ist VHDL in diesem Projekt effizienter, da Anforderungen präziser und mit geringerem Overhead umgesetzt und getestet werden konnten. HLS entfaltet seinen Vorteil primär bei rechenintensiven, algorithmischen Blöcken (z.,B. Filter, FFT) mit nennenswertem Parallelisierungspotenzial—nicht jedoch bei schlanker Low-Level-I/O-Logik.

6.2 Signal Framing Unit

Die SFU wurde sowohl als softwarebasierte Implementierung auf dem ARM-Core als auch als hardwarebeschleunigte Variante in HLS umgesetzt. Das Ziel bestand darin, den Einfluss von HLS auf Latenz und Ressourcenauslastung zu untersuchen sowie die Eignung dieser Methode für signalverarbeitende Module zu bewerten.

6.2.1 Funktionale Validierung

Die SFU realisiert ein Downsampling des Audiosignals von 44,1 kHz auf 22,05 kHz, segmentiert das Signal anschließend in Fenster mit 2048 Punkten und 50 % Überlappung

und multipliziert jedes Segment mit einem Hanning-Fenster. Damit werden die Anforderungen DV-1 bis DV-4 vollständig erfüllt.

Für die softwarebasierte Prozessorvariante wurden Testdaten in einem Array erzeugt und direkt auf dem ARM-Core verarbeitet. Die Ausgaben wurden mit den erwarteten Ergebnissen verglichen, um die Korrektheit des Downsamplings, der Segmentierung und der Fensterung zu bestätigen.

Für die HLS-Variante wurde eine Testbench erstellt, die sowohl eine C-Simulation als auch eine Co-Simulation in Vivado ermöglicht. Die Testdaten wurden in MATLAB erzeugt und für beide Simulationen verwendet, wodurch eine konsistente Überprüfung der HLS-Implementierung gewährleistet wurde.

Beide Varianten lieferten identische Ergebnisse, sodass die funktionale Korrektheit und Konsistenz der Implementierungen gesichert ist.

6.2.2 Messergebnisse und Ressourcenauslastung

Für die Prozessorvariante wurde die Latenz pro Sample direkt auf dem ARM-Core ermittelt. Dabei lagen die Werte bei etwa $2,1\mu$s pro Sample, was deutlich unter der kritischen Grenze von 22μs liegt, bei der ein neues Sample erwartet wird.

Die HLS-Variante wurde mit der zuvor erstellten Testbench synthetisch überprüft und anschließend auf dem FPGA implementiert. Hierbei wurde die Latenz auf rund $0,3,\mu$s reduziert. Gleichzeitig wurde die Auslastung der FPGA-Ressourcen gemessen, wobei die HLS-Implementierung nur wenige DSP-Slices, LUTs, Flip-Flops und BRAM beanspruchte.

Die Ergebnisse dieser Messungen sind in Tabelle 6.2 zusammengefasst. Sie zeigen, dass beide Varianten die Anforderungen problemlos erfüllen, während die HLS-Version zusätzliche Reserven in der Verarbeitungsgeschwindigkeit bietet.

Tabelle 6.2: Vergleich der SFU-Implementierungen

Variante	Latenz [μs]	DSP Slices	LUT	FF	BRAM
Software (RPU)	2,1	–	–	–	–
HLS	0,3	3	799	222	2

6.2.3 Analyse

Die softwarebasierte Umsetzung auf dem ARM-Core zeigt bereits sehr niedrige Latenzzeiten von $2,1,\mu$s pro Segment. Dies liegt daran, dass der Algorithmus der SFU relativ einfach ist: Downsampling, Segmentierung und Hanning-Fensterung erfordern nur wenige Rechenoperationen pro Sample, sodass die Berechnungen auf dem Prozessor effizient in einem einzelnen Durchlauf ausgeführt werden können.

Die HLS-Variante wurde aus dem gleichen Code übernommen, wobei lediglich minimale Anpassungen notwendig waren, um die AXI-Stream-Schnittstelle und die Synthese in Vitis-HLS zu ermöglichen. Trotz fehlender spezieller Optimierungen (wie Loop-Unrolling oder Pipelining) erzielte die HLS-Implementierung eine drastisch reduzierte Latenz von $0,3,\mu$s. Dies ist auf die automatische Parallelisierung: Operationen, die im Prozessor sequentiell ausgeführt werden, können in HLS teilweise gleichzeitig verarbeitet werden, wodurch die effektive Rechenzeit pro Segment sinkt.

Die Ressourcenauslastung der HLS-Variante bleibt moderat, da die SFU nur wenige Rechenoperationen und Speicherzugriffe erfordert. Die Analyse zeigt damit, dass die einfache Natur der SFU den ARM-Core bereits weit unter der kritischen Schwelle arbeiten lässt. HLS ermöglicht zusätzliche Beschleunigung, ohne signifikante Optimierung des Codes, um die Latenz zu reduzieren. Der Vorteil der HLS-Variante liegt vor allem in der parallelen Ausführung auf dem FPGA, während die Prozessorvariante ausreichend flexibel und leistungsfähig für den gegebenen Anwendungsfall ist.

Insgesamt verdeutlicht dies, dass für einfache Vorverarbeitungsschritte wie die der SFU eine reine Prozessorimplementierung bereits effizient und ausreichend ist, während HLS lediglich zusätzliche Leistungsreserven bereitstellt.

6.3 Mel Processing Unit

Die MPU wurde in zwei Varianten umgesetzt: einer Standard-HLS-Version und einer optimierten Variante mit Real-valued-FFT und sparsifizierter Mel-Filterbank. Das Ziel bestand darin, den Einfluss eines Optimierten HLS-Codes auf die Latenz und die Ressourcenauslastung zu untersuchen.

6.3.1 Funktionale Validierung

Die MPU berechnet das Mel-Spektrum der gefensterten Signale unter Verwendung einer FFT, einer Mel-Filterbank und einer logarithmischen Skalierung. Damit werden die Anforderungen DV-5 bis DV-7 vollständig erfüllt. Die Ergebnisse der Standard-HLS-Version und der optimierten Variante waren identisch, sodass die Korrektheit und Konsistenz beider Implementierungen bestätigt werden kann.

Zur Validierung wurde eine Testbench für HLS erstellt, die die MPU mit zuvor erzeugten Testdaten versorgt. Die Testdaten wurden mithilfe eines Python-Skripts erzeugt, das bekannte Audiosignale in Frames zerlegt und die erwarteten Mel-Spektren berechnet. Anschließend wurde das von der MPU erzeugte Spektrogramm eingelesen und mit den Referenzwerten aus dem Python-Skript verglichen.

6.3.2 Messergebnisse und Ressourcenauslastung

Die Latenzzeiten der beiden MPU-Varianten wurden mithilfe eines ILA auf dem FPGA gemessen, während die FPGA-Ressourcenauslastung aus der Vivado-Implementierung des vollständigen Designs ermittelt wurde. Die Ergebnisse sind in Tabelle 6.3 zusammengefasst.

Tabelle 6.3: Vergleich der MPU-Implementierungen

Variante	Latenz [μs]	DSP Slices	LUT	FF	BRAM
HLS Standard	169,03	20	4737	5364	14
HLS Optimiert	86,39	43	5905	6386	14

6.3.3 Analyse

Die Messergebnisse zeigen, dass die optimierte MPU-Variante mit Real-Valued-FFT und sparsifizierter Mel-Filterbank die Latenz im Vergleich zur Standard-HLS-Version nahezu halbieren konnte: von $169,03\,\mu$s auf $86,39\,\mu$s pro Frame. Dieser Effekt resultiert aus der effizienteren Berechnung der FFT für reelle Signale und der Reduktion redundanter Multiplikationen in der Filterbank.

Die Ressourcenauslastung stieg dabei insgesamt an: Die Anzahl der DSP-Slices erhöhte sich von 20 auf 43, und auch LUTs sowie Flip-Flops nahmen zu, da die optimierte Implementierung zusätzliche Logik für die Real-Valued-FFT, die Nachbearbeitung der FFT-Ausgänge sowie die skalierte Nutzung überlappender Filteranteile benötigt. Die BRAM-Nutzung blieb unverändert.

Die Validierung erfolgte über eine HLS-Testbench mit zuvor in Python generierten Referenzdaten sowie über ILA-Messungen auf dem FPGA. Hierbei zeigte sich, dass beide Varianten die funktionalen Anforderungen (DV-5 bis DV-7) erfüllen und die erzeugten Mel-Spektren identisch sind.

Insgesamt verdeutlicht die Analyse, dass sich gezielte algorithmische Optimierungen in HLS signifikant auf die Latenz auswirken können, allerdings auf Kosten einer höheren Ressourcenauslastung. Für zeitkritische Anwendungen, in denen Frames schnell verarbeitet werden müssen, bietet die optimierte Variante dennoch einen klaren Vorteil.

6.4 Audio Classification Unit

Die Audio Classification Unit führt die CNN-Inferenz vollständig in Software auf dem ARM-Core des Kria KV260 aus. Dabei kommt die PYNQ-Umgebung zum Einsatz. Eine geplante HLS-beschleunigte Hardwareimplementierung konnte aus Zeitgründen nicht mehr umgesetzt werden. Der Fokus wurde stattdessen darauf gelegt, eine stabile und lauffähige Implementierung bereitzustellen, um die Funktionalität des Gesamtsystems sicherzustellen.

6.4.1 Funktionale Validierung

Die ACU führt die CNN-Inferenz vollständig auf dem ARM-Core des Kria KV260 aus. Dazu wurde das trainierte CNN in der PYNQ-Umgebung innerhalb eines Jupyter Notebooks geladen. Zur Überprüfung der Korrektheit wurde der Validierungsdatensatz verwendet.

Die erzielten Klassifikationsergebnisse auf dem Kria KV260 waren identisch mit denen der Desktop-Referenzimplementierung. Die Klassifikationsgenauigkeit betrug 94%, womit die Anforderungen vollständig erfüllt werden. Dieses Ergebnis war erwartungsgemäß,

da das Netzwerk unverändert übernommen wurde und die Softwareumgebung dieselben numerischen Berechnungen wie die Referenz ausführt.

Die Validierung erfolgte durch den direkten Vergleich der Vorhersagen mit den erwarteten Labels aus dem Validierungsdatensatz, wodurch die funktionale Korrektheit und Stabilität der ACU bestätigt wurde.

6.4.2 Messergebnisse und Ressourcenauslastung

Die Ausführung der CNN-Inferenz auf dem ARM-Core des Kria KV260 wurde in Python innerhalb des Jupyter Notebooks gemessen. Dazu kam die Bibliothek `datetime` zum Einsatz, um die Laufzeiten präzise zu erfassen. Es zeigte sich, dass die Latenzen bei den ersten Klassifizierungen noch stark schwanken. Nach etwa zehn Durchläufen stabilisierte sich die Verarbeitung auf rund 400 ms pro Klassifizierung.

6.4.3 Analyse

Die initialen Schwankungen in der Latenz lassen sich durch den Start- und Ladeprozess des Netzwerks sowie die Initialisierung von PYNQ und Python-Runtime erklären. Beim ersten Durchlauf müssen sowohl das Modell als auch die Eingabedatenstrukturen vollständig in den Speicher geladen und die Interpreterumgebung initialisiert werden, was zu deutlich höheren Laufzeiten führt.

Nach mehreren Durchläufen stabilisieren sich die Latenzen bei etwa 400 ms pro Klassifizierung. Dies entspricht dem zu erwartenden Wert für die unbeschleunigte Softwareausführung auf dem ARM-Core des Kria KV260. Die Messungen zeigen, dass das System nach der Initialisierung eine konstante und reproduzierbare Verarbeitungsgeschwindigkeit liefert.

Zudem erwies sich die Implementierung in PYNQ als sehr unkompliziert und effizient, da das vortrainierte Netzwerk direkt in einem Jupyter Notebook geladen und ausgeführt werden konnte, ohne dass aufwendige Anpassungen oder zusätzliche Schnittstellenprogrammierung notwendig waren.

6.5 Vollständige Datenverarbeitung

Abschließend wurden alle Module - AIU, SFU, MPU und ACU - integriert und als vollständige Verarbeitungskette auf dem Kria KV260 SoC getestet. Ziel war es, die Gesamtlatenz zu ermitteln, die Ressourcenauslastung zu bewerten und sicherzustellen, dass alle Anforderungen H-1 bis H-8 sowie DV-1 bis DV-10 erfüllt werden.

6.5.1 Funktionale Validierung

Die vollständige Verarbeitungskette aus AIU, SFU, MPU und ACU arbeitete stabil und zuverlässig. Zur Überprüfung der korrekten Datenflüsse zwischen den Modulen wurde ein ILA eingesetzt. Dieser überwachte die AXI-Stream-Signale zwischen den Modulen und ermöglichte die Kontrolle der übermittelten Daten.

- Die AIU lieferte kontinuierlich die Audiodaten über AXI-Stream, wobei Taktung und Synchronisation der 16-Bit-Daten überprüft wurden.
- Die SFU führte Downsampling, Segmentierung und Hanning-Fenstermultiplikation korrekt aus, was anhand der ILA-Ausgabe zwischen AIU und SFU nachvollzogen werden konnte.
- Die MPU erzeugte das Mel-Spektrum und wendete FFT, Mel-Filterbank sowie logarithmische Skalierung fehlerfrei an; die Datenübergabe zwischen SFU und MPU wurde direkt über die ILA überprüft.
- Die ACU erhielt die Spektren korrekt, führte die Klassifikation aus und lieferte die Wahrscheinlichkeiten für alle zehn Klassen.

Zusätzlich wurden die Zeitpunkte der Datenübertragung und die Latenzen zwischen den Modulen mithilfe der ILA analysiert, um sicherzustellen, dass die Verarbeitung innerhalb der vorgegebenen Zeitfenster erfolgt. Die funktionale Validierung zeigte keine Signalverluste oder Synchronisationsprobleme und bestätigte die Einhaltung der Anforderungen.

6.5.2 Messergebnisse und Ressourcenauslastung

Die Ressourcenauslastung des Gesamtsystems wurde anhand der vollständigen Implementierung in Vivado bestimmt. Dabei wurden die Synthese- und Implementierungsreports genutzt, um die Belegung von DSP-Slices, LUTs, Flip-Flops und Block-RAM zu ermitteln.

Die Latenz zwischen den Modulen wurde mit dem ILA gemessen. Die ILA überwachte die AXI-Stream-Signale zwischen den Modulen und erlaubte die exakte Ermittlung der Verarbeitungszeit für ein 4-Sekunden-Audiosegment von der AIU bis zur Ausgabe der ACU.

Die Messergebnisse für die vier Varianten des Gesamtsystems sind in Tabelle 6.4 zusammengefasst:

Tabelle 6.4: Ressourcen der verschiedenen Implementierungen

Variante	DSP Slices	LUT	FF	BRAM
Variante 1	23	7983	9300	17
Variante 2	23	8017	9298	17
Variante 3	46	9104	10261	17
Variante 4	46	9142	10251	17

Die Gesamtlatenz für ein 4-Sekunden-Audiosegment lag bei etwa 410 ms, wobei die ACU den größten Anteil der Verarbeitungszeit beanspruchte. Ohne die ACU betrug die Latenz für die Varianten 1 und 2 rund 172 μs und für die Varianten 3 und 4 etwa 90 μs. Die Datenströme zwischen den Modulen blieben durchgehend stabil, und es konnten keine Signalverluste festgestellt werden.

6.5.3 Abschließende Analyse

Die Integration aller Module zu einer vollständigen Verarbeitungskette zeigt, dass das System sowohl funktional als auch leistungsmäßig den Anforderungen entspricht. Die ACU dominiert erwartungsgemäß die Gesamtlatenz, während die AIU, SFU und MPU mit sehr niedrigen Latenzen arbeiten – insbesondere die optimierte MPU-Variante, deren Latenz ohne ACU nur 90 μs beträgt.

Die Ressourcenauslastung liegt in allen Varianten im moderaten Bereich, sodass ausreichend Reserven für zukünftige Erweiterungen oder algorithmische Optimierungen bestehen. Die Überwachung der Datenströme mittels ILA bestätigte eine stabile und korrekte Verarbeitung zwischen allen Modulen, ohne Signalverluste oder Timing-Probleme.

Die Ergebnisse verdeutlichen, dass sich Hardware-Software-Co-Design und gezielte HLS-Optimierungen insbesondere bei rechenintensiven Modulen wie der MPU lohnen. Für die SFU hat sich gezeigt, dass eine direkte Übernahme des Prozessorcodes in HLS mit minimalen Anpassungen möglich ist, ohne zusätzliche Optimierungen vorzunehmen, da die Latenz ohnehin weit unter der kritischen Grenze liegt.

7 Fazit und Ausblick

Das Ziel dieser Arbeit bestand darin, den praktischen Nutzen von HLS bei der Entwicklung eingebetteter KI-Systeme zu bewerten und auf dieser Grundlage fundierte Empfehlungen für zukünftige Projekte abzuleiten.

Die Ergebnisse zeichnen ein klares Bild: HLS eignet sich besonders gut für abstrakte, datenflussorientierte Verarbeitungsschritte, wie sie beispielsweise in der digitalen Signalverarbeitung erforderlich sind. Aufgrund des höheren Abstraktionsgrads lassen sich Entwicklungszyklen verkürzen, Varianten schneller evaluieren und Funktionen leichter anpassen. Die in dieser Arbeit realisierten Hardwaremodule zur Berechnung von Mel-Spektrogrammen belegen, dass eine effiziente Umsetzung möglich ist, ohne dass es zu erheblichen Leistungseinbußen kommt.

Gleichzeitig wurden jedoch auch deutliche Grenzen sichtbar: Für hardware-nahe Aufgaben wie die Implementierung von Low-Level-Protokollen (z. B. I2S-Audioeingang) ist HLS kaum geeignet. Hier ist eine exakte Kontrolle über Signalflüsse, Taktzyklen und Schnittstellen erforderlich, die sich aus einer Hochsprachenbeschreibung nur unzureichend ableiten lässt. Solche Module erfordern weiterhin klassische Hardwarebeschreibungssprachen (z. B. VHDL oder Verilog), um eine präzise Implementierung sicherzustellen.

Aus der Kombination dieser Erkenntnisse lässt sich eine klare Empfehlung ableiten: HLS sollte gezielt für rechenintensive, algorithmische Module eingesetzt werden, während hardwarenahe Schnittstellen in HDL umgesetzt bleiben. Dieses differenzierte Vorgehen verbindet die Vorteile beider Ansätze: hohe Entwicklungsgeschwindigkeit durch HLS einerseits und maximale Kontrolle durch klassische HDL andererseits.

Darüber hinaus hat die Arbeit gezeigt, dass die Integration mehrerer Implementierungsvarianten – sowohl in Software auf ARM-Prozessoren als auch in Hardware auf FPGA-Logik – ein wirksames Mittel ist, um Flexibilität und Performance optimal auszubalancieren. Ein Vergleich der verschiedenen Varianten hat den technischen Wert der HLS-

Methodik bestätigt und die praktischen Trade-offs in Bezug auf Latenz, Ressourcenauslastung und Entwicklungsaufwand aufgezeigt.

7.1 Ausblick

Die in dieser Arbeit gewonnenen Erkenntnisse zum Einsatz der High-Level-Synthese im Hardware-Software-Co-Design bieten mehrere konkrete Ansatzpunkte für die Weiterentwicklung des Systems zur Vogelstimmenerkennung.

- **Zielgerichtete Erweiterung der Hardwarebeschleunigung:**
  Die Ergebnisse zeigen, dass HLS bei abstrakten Signalverarbeitungsaufgaben große Vorteile bietet. In zukünftigen Arbeiten könnten weitere rechenintensive Verarbeitungsschritte, etwa Teile der CNN-Inferenz, in die FPGA-Logik verschoben werden, um die Latenz weiter zu verringern und die Echtzeitfähigkeit auszubauen.

- **Feinabstimmung der Hardware-Software-Partitionierung:**
  Die Arbeit verdeutlicht, dass Low-Level-Module (z. B. I2S-Schnittstellen) weiterhin in VHDL implementiert werden sollten, während sich höhere Verarbeitungsebenen für eine effiziente Umsetzung mit HLS eignen. Eine systematische Evaluierung weiterer Varianten könnte dabei helfen, die optimale Aufgabenverteilung zwischen Prozessoren und Logik genauer zu bestimmen.

- **Validierung unter realen Einsatzbedingungen:**
  Das aktuelle System wurde überwiegend mit vorbereiteten Audiodaten getestet. Als nächster Schritt wäre der Dauerbetrieb mit kontinuierlichen Eingangssignalen aus realen Aufnahmesituationen vorgesehen. So könnten die Robustheit und die Zuverlässigkeit des Gesamtsystems geprüft sowie mögliche Schwachstellen in der Datenverarbeitung frühzeitig identifiziert werden.

Somit liefert diese Arbeit eine langfristig nutzbare, praxisorientierte Grundlage für den gezielten Einsatz von HLS in eingebetteten KI-Anwendungen. HLS ist ein leistungsfähiges Werkzeug, das an den richtigen Stellen eingesetzt werden sollte, also dort, wo Abstraktion und algorithmischer Fokus dominieren. Kritische, hardwarenahe Komponenten sollten dagegen besser in klassischer HDL verbleiben. Durch dieses differenzierte Vorgehen lässt sich die Entwicklungszeit verkürzen und ein robustes, echtzeitfähiges Gesamtsystem erreichen.

Literaturverzeichnis

[1] : *Getting Started — Python productivity for Zynq (Pynq).* – URL https://pynq.readthedocs.io/en/latest/getting_started.html. – Zugriffsdatum: 2025-08-15

[2] Ha, Soonhoi (Hrsg.) ; Teich, Jürgen (Hrsg.): *Handbook of Hardware/Software Codesign.* – URL http://link.springer.com/10.1007/978-94-017-7267-9. – Zugriffsdatum: 2025-09-02

[3] : *Kria K26 SOM Data Sheet (DS987).* – URL https://docs.amd.com/r/en-US/ds987-k26-som/Overview

[4] : *librosa.feature.melspectrogram — librosa 0.11.0 documentation.* – URL https://librosa.org/doc/0.11.0/generated/librosa.feature.melspectrogram.html#librosa.feature.melspectrogram. – Zugriffsdatum: 2025-08-24

[5] : *Pmod I2S2 Reference Manual - Digilent Reference.* – URL https://digilent.com/reference/pmod/pmodi2s2/reference-manual

[6] : *Practical Cryptography.* – URL http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/. – Zugriffsdatum: 2025-09-01

[7] : *PYNQ.* – URL http://www.pynq.io/. – Zugriffsdatum: 2025-08-15

[8] A, Sithara ; Thomas, Abraham ; Mathew, Dominic: Study of MFCC and IHC Feature Extraction Methods With Probabilistic Acoustic Models for Speaker Biometric Applications. 143, S. 267–276. – URL https://www.sciencedirect.com/science/article/pii/S1877050918320921. – Zugriffsdatum: 2024-12-08. – ISSN 1877-0509

[9] AMD Xilinx: *AMD Zynq^{TM} UltraScale+^{TM} MPSoCs.* – URL https://www.amd.com/de/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html. – Zugriffsdatum: 2025-09-03

[10] AMD Xilinx: *AXI DMA LogiCORE IP Product Guide (PG021)*

[11] AMD Xilinx: *AXI4-Stream Video IP and System Design Guide (UG934)*

[12] AMD Xilinx: *Fast Fourier Transform LogiCORE IP Product Guide (PG109)*

[13] AMD Xilinx: *PetaLinux Tools Documentation: Reference Guide (UG1144)*

[14] AMD Xilinx: *Vitis High-Level Synthesis User Guide (UG1399)*

[15] AMD Xilinx: ZCU104 Evaluation Board User Guide (UG1267).

[16] Anastasiia Purtova ; Indara Yanet Millan Lopez ; Kamila Shirinova: *Birds Singing Classification Using CNN Modelling Based On Mel-Spectrograms*

[17] ARM: *AMBA 4 AXI4-Stream Protocol Specification*

[18] Bahoura, Mohammed ; Ezzaidi, Hassan: *Hardware implementation of MFCC feature extraction for respiratory sounds analysis*

[19] Casseau, Emmanuel ; Le Gal, Bertrand: High-level synthesis for the design of FPGA-based signal processing systems. In: *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*, IEEE, S. 25–32. – URL http://ieeexplore.ieee.org/document/5289238/. – Zugriffsdatum: 2025-09-03. – ISBN 978-1-4244-4502-8

[20] Chanane, Hassen ; Bahoura, Mohammed: Convolutional Neural Network-based Model for Lung Sounds Classification. In: *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, URL https://ieeexplore.ieee.org/document/9531887/?arnumber=9531887. – Zugriffsdatum: 2024-12-24, S. 555–558. – ISSN: 1558-3899

[21] Cheng, Yu-Huei ; Chang, Pang-Ching ; Kuo, Che-Nan: Convolutional Neural Networks Approach for Music Genre Classification. In: *2020 International Symposium on Computer, Consumer and Control (IS3C)*, IEEE, S. 399–403. – URL https://ieeexplore.ieee.org/document/9394067/. – Zugriffsdatum: 2025-08-31. – ISBN 978-1-7281-9362-5

[22] Chien, Chiang-Heng ; Chien, Chiang-Ju ; Hsu, Chen-Chien: HW/SW Co-design and FPGA Acceleration of a Feature-Based Visual Odometry. In: *2019 4th International Conference on Robotics and Automation Engineering (ICRAE)*, IEEE, S. 148–152. – URL https://ieeexplore.ieee.org/document/9043811/. – Zugriffsdatum: 2025-09-02. – ISBN 978-1-7281-4740-6

[23] Choo, Chang ; Chang, Young-Uk ; Moon, Il-Young: FPGA-Based Hardware Accelerator for Feature Extraction in Automatic Speech Recognition. 13, Nr. 3, S. 145–151. – URL http://koreascience.or.kr/journal/view.jsp?kj=E1ICAW&py=2015&vnc=v13n3&sp=145. – Zugriffsdatum: 2024-12-20. – ISSN 2234-8255

[24] Digilent: Genesys ZU Reference Manual.

[25] Ehkan, Phaklen ; Zakaria, Fazrul F. ; Warip, MNM ; Sauli, Zaliman ; Elshaikh, Mohamed: Hardware Implementation of MFCC-Based Feature Extraction for Speaker Recognition. 339, S. 471–480. – ISSN 978-3-319-07673-7

[26] Elgendy, Mohamed: *Deep learning for vision systems*. Manning. – ISBN 978-1-61729-619-2 978-1-63835-041-5

[27] Elharrouss, Omar ; Akbari, Younes ; Almadeed, Noor ; Al-Maadeed, Somaya: Backbones-review: Feature extractor networks for deep learning and deep reinforcement learning approaches in computer vision. 53, S. 100645. – URL https://linkinghub.elsevier.com/retrieve/pii/S1574013724000297. – Zugriffsdatum: 2025-09-03. – ISSN 15740137

[28] Herber, Paula ; Glesner, Sabine: A HW/SW co-verification framework for SystemC. 12, Nr. 1, S. 1–23. – URL https://dl.acm.org/doi/10.1145/2435227.2435257. – Zugriffsdatum: 2025-09-02. – ISSN 1539-9087, 1558-3465

[29] Jiqing, Luo ; Husheng, Fang ; Qin, Yin ; Chunhua, Zhou: Quad-rotor UAV Audio Recognition Based on Mel Spectrum with Binaural Representation and CNN. In: *2021 International Conference on Computer Engineering and Application (ICCEA)*, IEEE, S. 285–290. – URL https://ieeexplore.ieee.org/document/9581062/. – Zugriffsdatum: 2025-08-31. – ISBN 978-1-6654-2616-9

[30] Knut Pröpper: *GitHub: MeisterProepper/Masterarbeit*. – URL https://github.com/MeisterProepper/Masterarbeit

[31] Kurniadhani, Bayuaji ; Hadiyoso, Sugondo ; Aulia, Suci ; Magdalena, Rita: FPGA-based implementation of speech recognition for robocar control using MFCC. 17, Nr. 4, S. 1914–1922. – URL https://telkomnika.uad.ac.id/index.php/TELKOMNIKA/article/view/12615. – Zugriffsdatum: 2025-01-06. – Number: 4. – ISSN 2302-9293

[32] LOUISE, RAMSAY CROCKETT H. .: *EXPLORING ZYNQ MPSOC: with pynq and machine learning applications*. STRATHCLYDE ACADEMIC MEDI. – OCLC: 1098337357. – ISBN 978-0-9929787-6-1

[33] Mertins, Alfred: *Signaltheorie: Grundlagen der Signalbeschreibung, Filterbänke, Wavelets, Zeit-Frequenz-Analyse, Parameter- und Signalschätzung*. 5. Auflage. Springer Vieweg. – ISBN 978-3-658-41528-0 978-3-658-41529-7

[34] Philips Semiconductors: *I2S bus specification*

[35] Reichardt, Jürgen: *Digitaltechnik und Digitale Systeme: Eine Einführung Mit VHDL*. 1st ed. Walter de Gruyter GmbH (De Gruyter Studium Series). – ISBN 978-3-11-070697-0

[36] Roberts, Leland: *Understanding the Mel Spectrogram*. – URL https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53. – Zugriffsdatum: 2025-08-31

[37] Schmidt, Erik ; West, Kris ; Kim, Youngmoo: *Efficient Acoustic Feature Extraction for Music Information Retrieval Using Programmable Gate Arrays.*. – Pages: 278

[38] Suhanek, Mia ; Petosić, Antonio ; Djurek, Ivan ; Slabbekoorn, Hans: Exploring the Influence of Avian Vocal Presence on Appraisal of Urban Soundscapes. 14, S. 11124

[39] Teich, Jürgen: Hardware/Software Codesign: The Past, the Present, and Predicting the Future. 100, S. 1411–1430. – URL https://ieeexplore.ieee.org/document/6172642/. – Zugriffsdatum: 2025-08-31. – ISSN 1558-2256

[40] Verboom, Willem: *Bird vocalizations: a female Common chaffinch song (Fringilla coelebs)*

[41] Verboom, Willem: *Bird vocalizations: songs of the European robin (Erithacus rubecula)*

[42] Verboom, Willem: *Bird vocalizations: the Great tit (Parus major)*

[43] Verboom, Willem: *Bird vocalizations: three Eurasian blue tit (Parus caeruleus) song components*

[44] Verboom, Willem ; Heij, Cornelis: *Bird vocalizations: song of the Eurasian blackcap (Sylvia atricapilla)*

[45] Verboom, Willem ; Heij, Cornelis: *Bird vocalizations: song repertoire of the Song thrush (Turdus philomelos)*

[46] Verboom, Willem C.: Bird vocalizations: songs of the Eurasian wren (Troglodytes troglodytes). . – URL http://rgdoi.net/10.13140/RG.2.2.22821.42726. – Zugriffsdatum: 2025-08-06. – Publisher: Unpublished

[47] Verboom, Willem C.: Drumming sound of the Great spotted woodpecker (Dendrocopos major). . – URL http://rgdoi.net/10.13140/RG.2.2.18567.09127. – Zugriffsdatum: 2025-08-11. – Publisher: Unpublished

[48] Verboom, Willem C. ; Heij, Cornelis J.: Bird vocalizations: Common chiffchaff (Phylloscopus collybita) songs. . – URL http://rgdoi.net/10.13140/RG.2.2.24778.80325. – Zugriffsdatum: 2025-08-11. – Publisher: Unpublished

[49] Wang, Jia-Ching ; Wang, Jhing-Fa ; Weng, Yu-Sheng: Chip design of MFCC extraction for speech recognition. 32, Nr. 1, S. 111–131. – URL https://linkinghub.elsevier.com/retrieve/pii/S0167926002000457. – Zugriffsdatum: 2024-12-19. – ISSN 01679260

[50] Willem-Pier Vellinga ; Robert Planqué ; Sander M. Pieterse: www.xeno-canto.org: a decade on.

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