Lehrdemonstrationen zur digitalen Filterung auf eingebetteten Systemen

Eduard Funkner Martin Schwarz

2025-07-28

Inhaltsverzeichnis

1. Motivation

Im Rahmen früherer Lehrveranstaltungen wurden bereits Erfahrungen mit analogen Filtern gesammelt. Auch digitale Filterstrukturen wurden theoretisch behandelt und mithilfe von MATLAB simuliert. Eine praktische Umsetzung digitaler Filter in realer Hardware blieb bisher jedoch aus. Daraus ergibt sich die Fragestellung, wie sich digitale Filter konkret in Hardware implementieren lassen. Dabei stellen sich zwei alternative Umsetzungswege dar, mit jeweils unterschiedlichen Anforderungen, Werkzeugen und Zielgruppen.

Dieses Repository vereint zwei eigenständige Arbeiten, die sich dieser Fragestellung auf unterschiedliche Weise nähern:

- Die erste Implementierung erfolgt in Arduino als auch in Micropython und nutzt das ESP-Lyrat Board als Zielplattform. Dieser Zugang richtet sich an Open-Source-Interessierte, Hobbyanwender und Bildungseinrichtungen. Die Umsetzung erfolgt vollständig mit kostenfreien Python-Tools sowie Bibliotheken und legt besonderen Wert auf Zugänglichkeit und Einfachheit.
- Die zweite Implementierung basiert auf einem FPGA, konkret dem PYNQ-Z2 Entwicklungsboard. Sie orientiert sich an industriellen Prozessen, verwendet MATLAB/Simulink in Kombination mit dem HDL Coder und zielt auf eine strukturierte Hardwareintegration ab. Ziel ist es, digitale Biquad-Filter als Vivado-kompatible IP-Cores umzusetzen und direkt auf dem PYNQ-Board demonstrieren zu können.

Beide Arbeiten stehen exemplarisch für unterschiedliche Realisierungsstrategien digitaler Signalverarbeitung, einmal industrieorientiert und einmal bildungsnah. Dabei sollen die Ergebnisse nicht nur die Machbarkeit demonstrieren, sondern auch als Einstiegshilfe für zukünftige Projekte und Lehre dienen.

Die digitale Signalverarbeitung ist ein zentraler Bestandteil moderner Anwendungen in der Kommunikations-, Audio- und Messtechnik. In vielen dieser Systeme müssen Signale gezielt gefiltert werden, um relevante Informationen zu extrahieren oder Störungen zu unterdrücken. Digitale Filter wie der Biquad-Filter sind dabei weit verbreitet, da sie mit vergleichsweise geringem Rechenaufwand komplexe Frequenzanpassungen ermöglichen.

Mit diesen Lehrdemonstrationen möchten wir zwei Ansätze zur Hardware-Implementierung näher erläutern. Dabei zeigen wir, wie ein digitaler Biquad-Filter auf beiden Plattformen umgesetzt wird und wie die jeweiligen Implementierungen funktionieren.

Part I.

Grundlagen

2. Theoritischen Grundlagen zur Implementierung von digitalen Biquads

3. Einführung

Diese Bachlorthesis dient als eine Lerndemonstration zur Implentierung von digitalen biquadratischen-Filtern auf Mikrocontrollern sowie FPGAs. Im Umfang dieser Lerndemonstration werden digitale Biquad-Filter mithilfe von Matlab- und Pythontools entworfen und im Anschluss auf Hard und Software umgesetzt. Der Prozess der Umsetzung wird zum Verständis dokumentiert sowie genau erklärt, damit Leser dieser Lerndemonstration mithilfe ihrer Hardware für sich den Prozess reproduzieren können.

4. Theoretische Grundlagen

Die nachfolgenden theoretischen Grundlagen dienen dem Verständnis der eingesetzten Konzepte und Verfahren der digitalen Signalverarbeitung, insbesondere im Kontext digitaler IIR-Filter.

Die Erklärungen stützen sich auf folgende Werke: - Applied Digital Signal Processing von Dimitris Manolakis und Vinay Ingle - Digital Signal Processing von John G. Proakis und Dimitris G. Manolakis - Digitale Signalverarbeitung von Karl-Dirk Kammeyer und Kristian Kroschel - Digitale Signalverarbeitung mit MATLAB von Martin Werner - Discrete-Time Signal Processing von Alan V. Oppenheim, Ronald W. Schafer und John R. Buck

Dieser Abschnitt entstand im Rahmen einer Zusammenarbeit mit einer weiteren Bachelorarbeit und liegt in beiden Arbeiten in identischer Form vor.

4.1. Infinite Impulse Response (IIR-Filter)

Infinite Impulse Response (IIR) Filter gehören in der Signalverarbeitung zu den grundlegenden Typen der digitalen Filter. IIR-Filter zeichnen sich fundamental durch ihre Charakteristik aus, bei der Berechnung des aktuellen Ausgangswertes durch die Verwendung des gegenwärtigen und vergangenen Eingabewertes als auch des vorherigen Ausgabewertes. Aufgrund dieser Rückkopplung kann es dazu führen, dass die Impulsantwort von IIR-Filtern theoretisch unendlich lang andauern kann, was sie grundsätzlich von Finite Impulse Response (FIR) Filtern unterscheidet.

Die theoretischen Grundlagen von IIR-Filtern basieren auf der Tatsache, dass die Filter sowohl Nullstellen als auch Polstellen besitzen. Während FIR-Filter ausschließlich Nullstellen aufweisen, ermöglichen die Polstellen von IIR-Filtern eine effizientere Umsetzung bestimmter Filtereigenschaften. Dies führt zu dem Hauptvorteil von IIR-Filter für die Möglichkeit scharfe Übergänge im Frequenzgang mit relativ wenigen Koeffizienten zu erreichen.

Ein immenser Vorteil von IIR-Filtern liegt in der hohen Effizienz bei der Realisierung scharfer Frequenzgänge mit relativ wenigen Koeffizienten, während FIR-Filter für vergleichbare Filtereigenschaften oft hunderte von Koeffizienten benötigen, können IIR-Filter ähnliche Ergebnisse mit deutlich geringerem Rechenaufwand erreichen. Aufgrund dieser Eigenschaft eignen sich IIR-Filter besonders gut im Einsatz von Systemen mit beschränkten Ressourcen oder in Echtzeitverarbeitung.

Biquad-Filter, kurz als “Biquad” bezeichnet, sind eine spezielle Unterklasse von IIR-Filtern zweiter Ordnung. Durch das Kaskadieren solcher Biquad-Sektionen besteht die Möglichkeit komplexe IIR-Filter höherer Ordnung auf einfache Weise zu realisieren. Das Kaskadieren dieser Sektionen bringt Vorteile in Bezug auf numerische Stabilität, Flexibilät bei der Parameteranpassung und Modularität des Designs.

Die Verwendung von Biquad-Sektionen anstelle von Filtern höherer Ordnung in direkter Form bietet eine Vielzahl von Vorteilen. Die Anfälligkeit für Koeffizientenquantisierung wird enorm gemindert, da die Pole und Nullstellen weniger stark von kleinen Änderungen der Koeffizienten abhängen. Im Weiteren ermöglicht die modulare Struktur eine unabhängige Optimierung jeder Sektion, wodurch der Entwurfsprozess vereinfacht wird.

4.2. Differenzengleichung eines IIR-Filters zweiter Ordnung

Digitale Filter können mit einer linearen Differenzengleichung beschrieben werden. Die biquadratischen Filter werden mit der Differenzengleichung eines IIR-Filters zweiter Ordnung beschrieben werden:

$$y[n] = \frac{1}{a_0}(b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2] - a_1 \cdot y[n-1] - a_2 \cdot y[n-2])$$

Der Wert $y[n]$ bezeichnet den Ausgabewert zum Sample n, während $x[n]$ den Eingangswert darstellt. Der aktuelle Ausgabewert $y[n]$ ergibt sich aus der gewichteten Summe der aktuellen und zwei vergangenen Eingabewerte, subtrahiert mit den zwei gewichteten vergangen Ausgabewerte.

Die Koeffizienten b_0, b_1 und b_2 bestimmen den Einfluss des aktuellen und der vergangenen Eingabewerte (Feedforward), während a_1 und a_2 den Rückkopplungsanteil aus früheren Ausgabewerten (Feedback) modellieren. Im Allgemeinem wird der Koeffizient auf a_0 auf 1 normiert, was zu der vereinfachten Form führt:

$$y[n] = b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2] - a_1 \cdot y[n-1] - a_2 \cdot y[n-2]$$

Durch diese Normierung wird die praktische Implementierung vereinfacht, da ein Multiplikationsschritt entfällt.

4.3. Übertragungsfunktion eines IIR-Filters zweiter Ordnung

Zur Analyse des Frequenz- und Phasenverhaltens des Filters, wird die Differenzengleichung mittels der z-Transformation in den z-Bereich transformiert und daraus folgt die resultierende Übertragungsfunktion:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

Die rationale Funktion eines biquadratischen Filters beschreibt das Verhältnis zwischen dem Ausgang $Y(z)$ zu dem Eingang $X(z)$. Diese Gleichung stellt die normierte Form des Biquads dar, bei welcher der Nennerkoeffizient auf $a_0 = 1$ normiert ist. Für den Fall, dass $a_0 \neq 1$ beträgt müssen die Koeffizienten durch a_0 geteilt werden, um die Normalform zu erreichen.

Anhand der Übetragungsfunktion können die Pol- und Nullstellen des Filters analysiert werden, um die Stabilität des Biquad-Filters zu vergewissern. Dabei müssen alle Polstellen des Filters innerhalb des Einheitskreises der z-Ebene liegen, beziehungsweise der Betrag jedes Pols kleiner als 1 sein muss.

Befinden sich Pole außerhalb oder direkt auf dem Einheitskreis, kann es dazu kommen, dass das Ausgangssignal unkontrolliert schwingt und das System damit instabil wird. Nullstellen haben hingegen lediglich Einfluss auf die Frequenzcharakteristik und nicht direkt auf die Stabilität.

4.4. Stabilität und Pol-Nullstellen-Verteilung

Die Stabilität eines IIR-Filters ist ein kritischer Faktor, wechler für die Implementierung des Filters gegeben sein muss. Ein kausaler IIR-Filter ist genau dann stabil, wenn alle Polstellen seiner Übertragungsfunktion innerhalb des Einheitskreises der komplexen z-Ebene liegen. Mathematisch ausgedrückt muss für alle Polstellen $|p_i| < 1$.

Die Lage der Pole bestimmt nicht nur die Stabilität, sondern auch das dynamische Verhalten des Filters. Pole die näher am Rand des Einheitskreises liegen führen zu einem resonanten Verhalten mit langen ausschwingendem Verhalten, während Pole näher zum Ursprung hin, ein gedämpfteres Verhalten bewirken. Die Nullstellen hingegen beeinflussen primär den Verlauf des Frequenzgangs.

Für die praktische Implementierung ist es wichtig zu beachten, dass Quantisierungseffekte sich auf die Lage der Pole auswirken können. Das kann im schlimmsten Fall dazu führen, sodass die Stabilität des Filters beeinträchtigt und der Filter instabil wird. Aus diesem Grund ist bei der Koeffizientenquantisierung besondere Vorsicht geboten, bei Filtern mit Polen nah oder auf dem Rand des Einheitskreises.

4.5. Frequenzgang und Phasenverhalten

Der Frequenzgang eines IIR-Filters wird durch die Auswertung der Übertragungsfunktion innerhalb des Einheitskreises erhalten, dafür wird die Substitution $z = e^{j\omega}$ durchgeführt:

$$H(e^{j\omega}) = \frac{b_0 + b_1 e^{-j\omega} + b_2 e^{-2j\omega}}{1 + a_1 e^{-j\omega} + a_2 e^{-j2\omega}}$$

Der Betrag $|H(e^{j\omega})|$ beschreibt die Amplitudencharakteristik, während die Phase $\arg(H(e^{j\omega}))$ das Phasenverhalten wiedergibt. Bei IIR-Filtern ist das Phasenverhalten im Allgemeinen nicht linear, was in manchen Anwendungen als unerwünscht erweist.

5. Implementierungsstrukturen biquadratischer Filter

Die praktische Implementierung von biquadratischen Filtern erfolgt durch die direkte Umsetzung der Differezengleichungen in Software oder Hardware.

Dafür werden die verzögerten Eingangswerte $x[n-1]$ und $x[n-2]$, sowie die Ausgangswerte $y[n-1]$ und $y[n-2]$ in Speicherelementen, sogenannten Verzögerungsgliedern, gespeichert. Der aktuelle Ausgangswert $y[n]$ wird mittels der gewichteten Summation aller Terme gemäß der jeweiligen Differenzengleichung berechnet.

Bei der numerischen Implementierung sind verschiedene Aspekte zu berücksichtigen, insbesondere die Wortbreite und des Zahlenformats (Festkomma oder Gleitkomma).

Weitere Einflüsse wie Rundungsfehler oder Quantisierungsrauschen können die Filtercharakteristik beeinflussen und im Extremfall die Stabilität des Filters gefährden.

Diese Implementierungsstrukturen bieten die Möglichkeit, mehrere Filter-Sektionen aneinander zu Verbinden. Diese Modularität biquadratischer Filter ermöglicht, komplexere Filtersysteme höherer Ordnung, durch die Kaskadierung mehrerer Biquad-Sektionen zu realisieren.

Dadurch können Vorteile in Bezug auf numerische Stabilität, Designflexibilität und Implementierungseffizienz geschaffen werden, weil jede Sektion unabhängig entworfen und optimiert werden kann.

5.1. Direktform 1

Die Direktform 1 stellt die direkteste Umsetzung der biquadratischen Differenzengleichung in einer Signalflussstruktur dar.

Diese Implementierung folgt direkt der mathematischen Beschreibung unter der Verwendung separater Verzögerungsketten für die Eingangs- als auch die Ausgangswerte.

Diese Struktur kann als eine Parallelschaltung eines rein rekursiven Teils (nur Pole) und eines nicht rekursiven Teils (nur Nullstellen) aufgefasst werden.

$$y[n] = b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2] - a_1 \cdot y[n-1] - a_2 \cdot y[n-2]$$

Für die Direktform 1 Implementierung werden zunächst alle gewichteten Eingangswerte und deren Verzögerungen zusammengefasst und berechnet:

$$w[n] = b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2]$$

Anschließend wird die Rückkoppling der verzögerten Ausgangswerte realisiert:

$$y[n] = w[n] - a_1 \cdot y[n-1] - a_2 \cdot y[n-2]$$

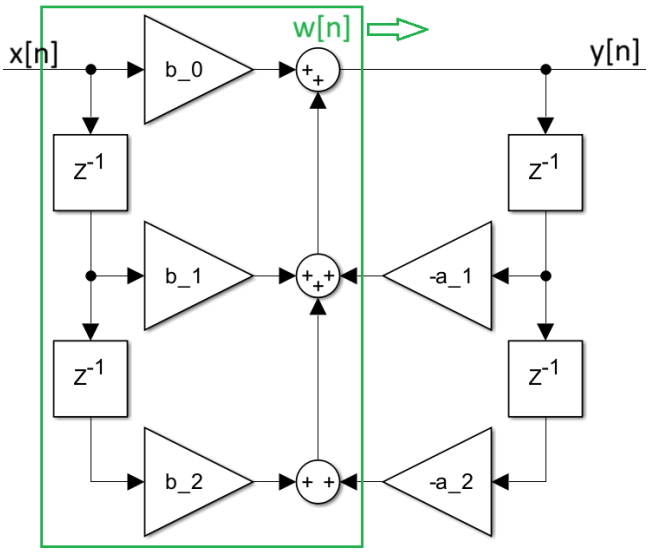


Abbildung 5.1.: Signalflussbild: Direktform 1

In dieser Struktur werden vier Verzögerungselemente verwendet: zwei für die Eingangsverzögerung, $x[n-1]$ und $x[n-2]$, sowie zwei für die Ausgangsverzögerung, $y[n-1]$ und $y[n-2]$. Die Multiplikationen zwischen den Verzögerungselementen und den Koeffizienten erfolgen parallel, wodurch sich eine hohe Verarbeitungsgeschwindigkeit erzielen lässt.

Ein wesentlicher Vorteil der Direktform 1 liegt in ihrer Robustheit gegenüber Koeffizientenquantisierung. Da die Feedforward- und Feedback-Pfade getrennt sind, beeinflussen Quantisierungsfehler in einem Pfad nicht direkt den anderen. Dies führt zu einer besseren numerischen Stabilität, besonders bei der Verwendung von Festkommaarithmetik .

Die Direktform 1 zeichnet sich durch ihre konzeptionelle Einfachheit und die direkte Korrespondenz zur Differenzengleichung aus. Nachteilhaft bei dieser Struktur ist der erhöhte Speicherbedarf, aufgrund der redundanten Verzögerungselemente. Für Anwendungen mit strengen Speicherbeschränkungen kann dies ein limitierender Faktor sein.

5.2. Direktform 2 (Kanonische Form)

Die Direktform 2, oder auch als kanonische Form bekannt, ist eine optimierte Variante der Direktform 1, welche durch die Umordnung der Operationen die Anzahl der benötigten Verzögerungselemente minimiert.

Diese Struktur macht sich die Kommutativität (Vertauschbarkeit) linearer zeitinvarianter (LTI)-Systeme zunutze, indem sie die Reihenfolge von der Vorwärts- und Rückwärtsverarbeitung vertauscht.

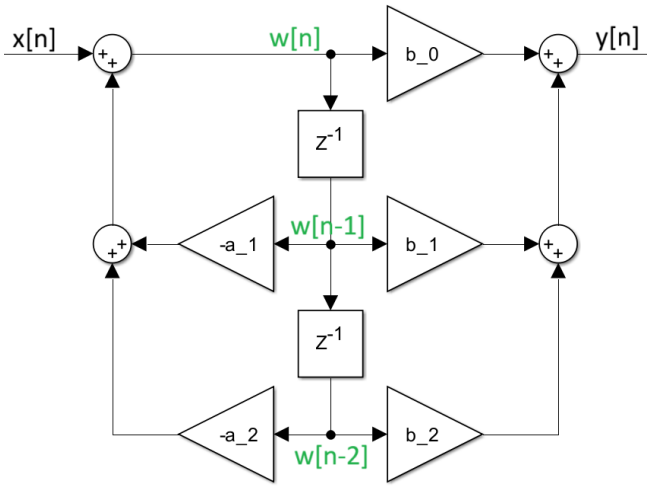


Abbildung 5.2.: Signalflussbild: Direktform 2

Zunächst wird ein Zwischenwert $w[n]$ durch die rekursive Beziehungen erzeugt:

$$w[n] = x[n] - a_1 \cdot w[n-1] - a_2 \cdot w[n-2]$$

Der Zwischenwert $w[n]$ dient als Substitution für die Verzögerungselemente der Ein- und Ausgangswerte.

$$y[n] = b_0 \cdot w[n] + b_1 \cdot w[n-1] + b_2 \cdot w[n-2]$$

In dieser Struktur werden nur zwei Verzögerungselemente für w[n-1] und w[n-2] benötigt, da die rekursive als auch die nicht-rekursive Verarbeitung auf dieselben verzögerten Werte zugreifen. Die Bezeichnung “kanonisch” bezieht sich darauf, dass diese Implementierung die minimal mögliche Anzahl von Verzögerungselementen verwendet und dadurch die Speichereffizienz optimiert.

Die Direktform 2 eignet sich Ideal für Systeme mit begrenzten Speicherressourcen. Der reduzierte Speicherbedarf ist insbesondere bei der Implementierung auf Mikrocontrollern oder in FPGA-Anwendungen von hohem Vorteil, bei welchen Speicher eine limitierte und kostbare Ressource darstellt.

Jedoch weist diese Struktur eine erhöhte Anfälligkeit gegenüber Quantisierungseffekten auf, da die Zwischenwerte $w[n]$ größere Dynamikbereiche aufweisen können als die ursprünglichen Ein- und Ausgangswerte. Dies kann zu einer Verschlechterung des Signal-Rausch-Verhältnisses führen, primär bei der Verwendung von Festkommaarithmetik mit begrenzter Wortbreite.

5.3. Transponierte Direktform 2

Die transponierte Direktform 2 kann gebildet werden durch die Anwendung des Transpositionstheorems auf die Direktform 2. Dieses Theorem besagt, dass die Umkehrung aller Signalflussrichtungen bei gleichzeitiger Vertauschung von Eingängen und Ausgängen eine äquivalente Übertragungsfunktion ergibt. Die resultierende Struktur weist oft günstigere numerische Eigenschaften auf, als die ursprüngliche Form.

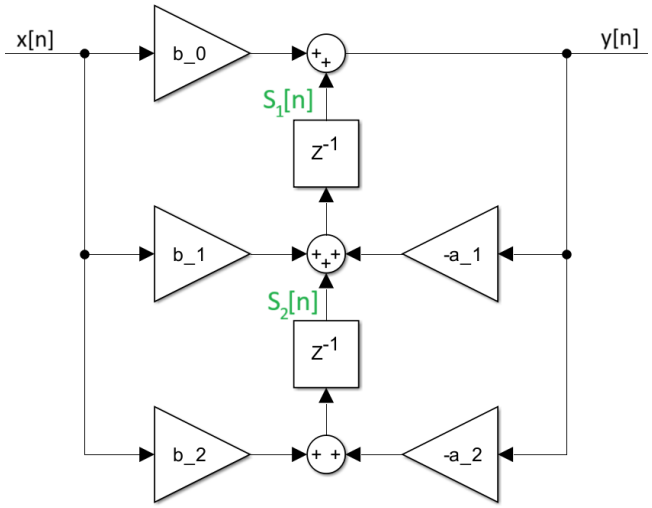


Abbildung 5.3.: Signalflussbild: Transponierte Direktform 2

Die transponierte Direktform 2 wird druch die folgenden Differenzengleichungen beschrieben:

$$s_2[n] = b_2 \cdot x[n] - a_2 \cdot y[n]$$

$$s_1[n] = s_2[n] + b_1 \cdot x[n] - a_1 \cdot y[n]$$

$$y[n] = s_1[n] + b_0 \cdot x[n]$$

Bei der transponierten Direktform 2 werden die Zwischenwerte $s_1[n]$ und $s_2[n]$ als Verzögerungselemente verwendet. Diese Struktur eignet sich besonders gut in der Implementierung von Echtzeitanwendung,

aufgrund ihrer effizienten Berechnung und der guten Eignung für parallele oder pipelined Hardware-Architekturen.

Ein wesentlicher Vorteil der transponierten Direktform 2 liegt in ihrer reduzierten Anfälligkeit gegenüber Koeffizientenquantisierung. Die Rückkopplungsschleifen sind kürzer als in der Direktform 2, was zu einer verbesserten numerischen Stabilität führt. Darüber hinaus ist die Struktur besonders gut für die Implementierung in Hardware geeignet, da die Berechnungen eine natürliche Pipeline-Struktur ausweisen.

6. Numerische Aspekte und Quantisierungseffekte

Die praktische Implementierung digitaler Filter auf Systemen mit endlicher Wortlänge bringt verschiedene Herausforderungen mit sich. Diese Effekte können die Filterleistung erheblich beeinträchtigen und müssen bereits in beim Entwerfen berücksichtigt werden. Die wichtigsten numerischen Aspekte sind Koeffizientenquantisierung, Quantisierungsrauschen(Rundungsrauschen) und nichtlineare Quantisierungseffekte.

6.1. Koeffizientenquantisierung

Die Koeffizientenquantisierung ist einer der kritischen Aspekte bei der Implementierung digitaler Filter auf Systemen mit begrenzter Wortlänge. Bei der Darstellung der Filterkoeffizienten mit einer endlichen Anzahl von Bits entstehen Quantisierungsfehler, die zu einer Verschiebung der Pole und Nullstellen des Filters führen kann.

Auswirkung der Koeffizientenquantisierung

Die Quantisierung der Koeffizienten führt zu verschiedenen unerwünschten Effekten:

Polverschiebung: Die quantisierten Koeffizienten verschieben die Pole des Filters, was zu einer Änderung der Filtercharakteristik führt. Im schlimmsten Fall können die Pole aus dem Einheitskreis herauswandern und zu einem instabilen Filter führen.

Frequenzgangverzerrung: Die Verschiebung der Pole und Nullstellen führt zur Abweichung im Frequenzgang. Besonders kritisch ist dies bei scharfen Filtern mit hoher Güte.

Stabilitätsverlust: Pole am oder auf dem Rand des Einheitskreises kann bereits durch kleine Quantisierung zur Instabilität führen. Die Wahrscheinlichkeit der Instabilität steigt mit abnehmender Wortlänge.

Strukturelle Gegenmaßnahmen

Die Wahl der Filterstruktur hat einen wesentlichen Einfluss auf die Robustheit gegenüber Quantisierungseffekten. Die Direktform 1 ist aufgrund der getrennten Pfade für Feedforward- und Feedbackanteile weniger anfällig für Fehlerausbreitung und bietet bessere numerische Stabilität bei Festkommaarithmetik. Die Direktform 2 bietet gegenüber der Direktform 1 eine bessere Speichereffizienz, jedoch können größere Zwischenwerte zum Überlaufen bei begrenzter Wortbreite mit erhöhtem Rauschen führen.

Die transponierte Direktform 2 kombiniert die Vorteile der beiden genannten Strukturen und zeichnet sich durch eine besonders hohe Resistenz gegenüber Koeffizientenquantisierung aus. Dies liegt an der

kürzeren Rückkopplungskette und der numerisch günstigen Struktur für Echtzeitanwendungen. Bei der Implementierung der digitalen Filter ist es zu empfehlen eine Wortlänge von mindestens 16 Bit mit enger Bandbreite zu wählen, um Koeffizientenquantisierung bei einem Minimum zu halten.

6.2. Quantisierungsrauschen (Rundungsrauschen)

Bei der Implementierung digitaler Filter entstehen bei jeder arithmetischen Operation Rundungsfehler, die sich als additives Rauschen am Filterausgang manifestieren. Diese Fehler sind unvermeidlich bei der Verwendung von Festkommaarithmetik und können das Signal-Rausch-Verhältnis des Filters erheblich verschlechtern. Um diesem Entgegen zu wirken, kann die durch die Wahl einer geeigneten Implementierungsstruktur das Rundungsrauschen minimiert werden.

Direktform 1: Die Rundungsrauschquellen in den Feedforward- und Feedback-Pfaden sind weitgehend entkoppelt. Dadurch führt es zu einer moderaten Gesamtvarianz des Rundungsrauschens. **Direktform 2**: Die gemeinsame Nutzung der Verzögerungselemente führt zu einer höheren Rundsrauschvarianz, da die Rundungsfehler mehrfach durch das rekursive System zirkulieren. **Transponierte Direktform 2**: Diese Struktur weist das günstigste Rundungsrauschen auf, da die Rundungsrauschquellen näher am Ausgang liegen und weniger stark gefiltert werden.

6.3. Limit-Cycle-Oszilationen

Limit-Cycle-Oszillationen sind selbsterhaltende Schwingungen, die in nichtlinearen digitalen Filtern auftreten können. Sie entstehen durch die Kombination von Quantisierung und Rückkopplung und können auch bei stabilen linearen Filtern auftreten, wenn nichtlineare Quantisierungseffekte berücksichtigt werden.

Zero-Input Limit Cycles sind Oszillationen die auftreten, wenn das Eingangssignal null entspricht, aber der Filter aufgrund gespeicherter Energie in den Verzögerungselementen weiterhin schwingt.

Large-Scale Limit können bei hohen Eingangssignalpegeln auftreten und sind nicht auf kleine Amplituden beschränkt. Sie entstehen durch die Sättigungscharakteristik der Arithmetik und können zu großen, störenden Ausgangssignalen führen.

6.4. Kaskadierung

Für Filter höherer Ordnung bietet die kaskadierte-Realisierung aus Biquad-Sektionen deutliche Vorteile gegenüber der direkten Implementierung. Die optimale Anordnung und Paarung der Pole und Nullstellen in einzelnen Sektionen kann die numerische Stabilität verbessern, sowie das Rundungsrauschen minimieren als auch die Anfälligkeit für Koeffizientenquantisierung reduzieren.

6.5. Strukturenvergleich

Eigenschaft	Direktform 1	Direktform 2	Transp. Direktform 2
Verzögerungselemente	4	2	2
Speicherbedarf	Hoch	Minimal	Minimal
Koeff.-Empfindlichkeit	Niedrig	Hoch	Mittel
Rundungsrauschen	Mittel	Hoch	Niedrig
Limit-Cycle-Anfälligkeit	Niedrig	Hoch	Niedrig
Hardware-Eignung	Gut	Mittel	Sehr gut
Dynamikbereich	Gut	Begrenzt	Sehr gut
Quantisierungsrobustheit	Hoch	Niedrig	Mittel

Rundungsrauschverhalten: Die transponierte Direktform 2 weist das günstigste Rundungsrauschverhalten auf, da die Rundungsquellen näher am Ausgang liegen und weniger stark durch die Rückkopplung verstärkt werden. Die Direktform 2 zeigt das ungünstigste Verhalten, da die Rundungsfehler durch die rekursive Struktur mehrfach zirkulieren. **Koeffizientenempfindlichkeit**: Die Direktform 1 bietet die beste Robustheit gegenüber Koeffizientenquantisierung, da die Feedforward- und Feedback-Koeffizienten unabhängig quantisiert werden können. Die Direktform 2 zeigt die höchste Empfindlichkeit aufgrund der gekoppelten Struktur.

Die Wahl der optimalen Struktur hängt von den spezifischen Anforderungen der Anwendung ab. Für Anwendungen mit strengen Speicherbeschränkungen sind die Direktform 2 und transponierte Direktform 2 vorzuziehen. Bei hohen Anforderungen an die numerische Genauigkeit und Robustheit ist die transponierte Direktform 2 oder die Direktform 1 die bessere Wahl. Für Echtzeitanwendungen eignet sich die transponierte Direktform 2 besonders gut aufgrund ihrer speicher- und rechen Zeiteffizienz sowie ihrer numerischen Stabilität.

7. Bilineartransformation

Die Bilineartransformation dient als ein Verfahren, welches die Überführung analoger kontinuierlicher Filter in digitale Filterstrukturen ermöglicht. Bei diesem Verfahren wird das analoge Frequenzverhalten möglichst genau in den digitalen Frequenzbereich übertragen und bietet dabei die Erhaltung der Stabilität sowie eine bijektive, aliasfreie Abbildung der s-Parameter in den z-Bereich. Sei überführt die imaginäre Achse der s-Ebene ($j\omega$) exakt auf den Einheitskreis der z-Ebene und erhält dabei die Stabilitätsbedingungen.

7.1. Zentrale Substitutionen

Die mathematische Basis der Bilineartransformation beruht auf einer Substitution der s Parameter:

$$s = \frac{2}{T_d} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}$$

Dabei ist T_d ein Entwurfsparameter mit Zeitcharakteristik, der in der Praxis überlicherweise mit der Abtastperiode T gleichgesetzt wird. Durch dieses nichtlineare Transformation werden rationale Funktionen im s-Bereich in rationale Funktionen im z-Bereich abgebildet, ohne dass die Orndung der Übertragungsfunktionen ansteigt.

7.2. Transformation der Übergangsfunktion

Zur Digitalisierung eines analogen Filters mit der Übertragungsfunktion $H_c(s)$ wird die Subtitution in die analoge Übertragungsfunktion eingesetzt. Dieser Prozess führt zur digitalen Übertragungsfunktion:

$$H(z) = H_c\left(\frac{2}{T_d} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}\right)$$

Die resultierende Übertragungsfunktion $H(z)$ repräsentiert die vollständige digitale Überführung des ursprünglichen analogen Filters, wobei alle wesentlichen Filtereigenschaften erhalten bleiben.

7.3. Transformation von Polen und Nullstellen

Die Pol- und Nullstellen des analogen Filters unterliegen während der Transformation spezifischen Abbildungsregeln. Für eine analoge Nullstelle ζ_k und einen analogen Pol s_k ergeben sich die entsprechenden digitalen Werte:

$$z_k = \frac{1 + \frac{T_d}{2} \cdot \zeta_k}{1 - \frac{T_d}{2} \cdot \zeta_k}, \quad p_k = \frac{1 + \frac{T_d}{2} \cdot s_k}{1 - \frac{T_d}{2} \cdot s_k}$$

Diese Transformationsregeln gewährleisten, dass die fundamentalen Eigenschaften des Filters während der Digitalierung erhalten bleiben. Insbesondere stellt die Transformation sicher, dass die linke s-Halbebene, welche alle stabilen Pole enthält, vollständig in das Innere des Einheitskreises der z-Ebene abgebildet werden kann.

7.4. Frequenzverzerrung - Frequency Warping:

Ein wesentlicher und unvermeidlciher Aspekt der Bilineartransformation ist die nichtlineare Abbildung der Frequenzachse, bekannt als Frequenzverzerrung oder Frequency Warping. Die mathematische Beschreibung der Frequenzformation lautet:

$$\omega = 2 \cdot \tan^{-1}\left(\frac{\Omega T_d}{2}\right), \quad \Omega = \frac{2}{T_d} \cdot \tan\left(\frac{\omega}{2}\right)$$

Durch die Frequenzverzerrung kann das Verhalten digitaler Filter beeinflusst werden. Bei niedrigen Frequenzen, nahe $\omega = 0$ oder 0 Hz, ist die Verzerrung minimal und nahezu vernachlässigbar. In diesem Bereich verhalten sich digitale und analoge Filter nahezu identisch. Mit steigender Frequenz nimmt die Verzerrung progressiv zu und erreicht ihr Maximum bei Frequenzen nahe der Nyquist-Frequenz $\omega = \pi$ oder der halben Abtastrate $\frac{f_s}{2}$

7.5. Prewarping

Um die Abweichungen die durch das Frequency Warping entstehen zu kompensieren, wird die Methode des Prewarpings genutzt. Dabei wird die analoge Designfrequenz Ω_k so angepasst, dass sie nach der Bilineartransformation genau der gewünschte digitalen Frequenz ω_k entspricht:

$$\Omega_k = \frac{2}{T_d} \cdot \tan\left(\frac{\omega_k}{2}\right) = 2 \cdot f_s \cdot \tan(\frac{\pi f_d}{f_s})$$

In der praktischen Anwendung wird Prewarping typischerweise für Eckfrequenzpunkte eingesetzt. Dazu gehören Grenzfrequenzen von Tief- und Hochpassfiltern, Mittenfrequenzen von Bandpassfiltern, sowie Punkte der -3dB-Frequenzen oder Grenzen von Sperrbereichen. Durch die Anwendung des

Prewarpings auf diese gezielten Punkte kann sichergestellt werden, dass der digitale Filter an den spezifizierten Frequenzen das gewünschte Verhalten aufweist.

Prewarping ist jedoch nicht zwingend notwendig, wenn die Abtastastrate f_s höher als die Eckfrequenzen f_d der Filter gewählt wird. Dadurch wird die Abbildung durch die Bilineartransformation nahezu linear und die Frequenzverzerrung ist vernachlässigbar gering.

Part II.

Mikrocontroller

8. Lerndemonstration zur Implementierung von digitalen Filtern auf Mikrocontrollern

9. Abstract

Diese Bachelorarbeit behandelt die praxisnahe Implementierung biquadratischer digitaler IIR-Filter (Biquads) auf Mikrocontrollerbasis anhand des Lyrat V4.3 mit dem ESP32. Ziel der Arbeit ist es, die theoretischen und praktischen Grundlagen digitaler Biquad-Filter zu vermitteln und in einer anwendungsorientierten Lerndemonstration umzusetzen. Dabei werden verschiedene Filterstrukturen wie Direktform 1, Direktform 2 sowie die transponierte Direktform 2 in Arduino und MicroPython realisiert und miteinander verglichen. Der Fokus liegt auf der Audioverarbeitung, insbesondere der Verarbeitung von WAV-Dateien sowie die Echtzeit-Filterung über Mikrofoneingänge.

Ein zentraler Bestandteil des Gesamtprozesses ist der digitale Filterentwurf in Python. Mithilfe von Bibliotheken wie scipy.signal sowie dem Tool pyfda werden Biquads entworfen, analysiert und deren Koeffizienten zur Implementierung auf dem Mikrocontroller exportiert. Diese Vorgehensweise ermöglicht eine effiziente Verifikation der Filtereigenschaften im Vorfeld und bildet die Brücke zwischen Design und Hardwareumsetzung.

Durch die Kombination von theoretischen Grundlagen, grafischen Designwerkzeugen, kommentierten Codebeispielen und praxisnahen Anwendungen bietet diese Arbeit eine kompakte Anleitung für Studierende und Entwickler, um eigene digitale Filterlösungen auf Mikrocontrollern mit begrenzten Ressourcen zu realisieren und weiterzuentwickeln. Die gesamten Inhalte dieser Arbeit sind innerhalb eines Git Repositorys hinterlegt und dokumentiert.

10. Motivation

Die Motivation dieser Arbeit liegt darin, die erlernten Grundkenntnisse aus der Theorie in die Praxis umzusetzen. Im Rahmen des Studiums wurden die Grundlagen der digitalen Signalverarbeitung, insbesondere digitaler Filter, intensiv behandelt, jedoch meist in abstrakter und simulierter Form. Ziel dieser Thesis ist es daher, diese theoretischen Konzepte durch eine praxisnahe Anwendung auf echter Hardware greifbar zu machen und so ein tieferes Verständnis für deren Umsetzung und Verhalten im realen System zu gewinnen.

Im Mittelpunkt der Arbeit steht die Implementierung digitaler Biquad-Filter auf dem Lyrat-Audioboard mit dem ESP32. Dabei werden verschiedene Filterstrukturen wie Direktform 1, Direktform 2 und die transponierte Direktform 2 implementiert und im Hinblick auf ihre Eigenschaften verglichen. Neben der Entwicklung der Filterlogik in C++ (Arduino) und MicroPython werden auch praxisrelevante Anwendungsbeispiele realisiert, darunter die Echtzeitfilterung über den Mikrofoneingang sowie die Verarbeitung von WAV-Dateien von einer SD-Karte.

Ergänzend erfolgt der digitale Filterentwurf mithilfe von Python-Tools wie scipy.signal und pyfda, wodurch die benötigten Koeffizienten grafisch analysiert und direkt für die Implementierung aufbereitet werden können. Durch diese Verbindung aus Theorie, Software-Design und Hardwareumsetzung entsteht eine vollständige Lerndemonstration, die nicht nur den eigenen Lernfortschritt fördert, sondern auch als Hilfestellung für andere Studierende und Entwickler dienen kann, die sich mit digitaler Signalverarbeitung auf Mikrocontrollerbasis beschäftigen.

11. Einführung

Die digitale Signalverarbeitung ist ein zentraler Bestandteil moderner Audio- und Kommunikationstechnik. Insbesondere digitale Filter spielen eine entscheidende Rolle bei der Rauschunterdrückung, Signalglättung und Frequenzselektion. In ressourcenbeschränkten eingebetteten Systemen wie Mikrocontrollern müssen solche Filter effizient, stabil und echtzeitfähig implementiert werden. Eine weit verbreitete Struktur sind dabei Biquad-Filter, die sich durch geringe Rechenlast, gute Skalierbarkeit und hohe Flexibilität auszeichnen.

Ziel dieser Bachelorarbeit ist die systematische Entwicklung, Berechnung und Umsetzung digitaler Biquad-Filter auf dem Lyrat-Audioboard mit dem ESP32. Als praxisorientierte Lerndemonstration wird gezeigt, wie digitale Filter mit den verschiedenen Strukturen: Direktform 1, Direktform 2 und transponierte Direktform 2, auf Mikrocontrollern programmiert und eingesetzt werden können. Die Implementierung erfolgt in zwei Programmierumgebungen: Arduino und MicroPython. Im Vordergrund stehen dabei sowohl die Filterung von WAV-Dateien als auch die Echtzeitverarbeitung von Audiosignalen über den Mikrofoneingang.

Ein wichtiger Bestandteil des Workflows ist der digitale Filterentwurf in Python. Mit Hilfe von Bibliotheken wie SciPy, NumPy und dem Tool pyFDA werden die benötigten IIR-Koeffizienten entworfen, analysiert und für die spätere Implementierung auf dem Mikrocontroller vorbereitet. Durch die grafische Analyse lassen sich Frequenzgang, Stabilität und Quantisierungseinflüsse bereits im Vorfeld bewerten und optimieren. Der Entwurf in Python dient dabei als flexible und zugängliche Plattform zur Verifikation und Visualisierung der Filtereigenschaften, bevor die praktische Umsetzung auf dem Zielsystem erfolgt.

Die Arbeit richtet sich an Studierende und Entwickler, die sich mit digitaler Signalverarbeitung in eingebetteten Systemen beschäftigen. Sie kombiniert theoretische Grundlagen, praxisorientierte Entwurfswerkzeuge und kommentierte Codebeispiele, um einen umfassenden und anwendungsnahen Einstieg in die Implementierung digitaler Filter auf Mikrocontrollern zu bieten.

Diese Bachelorarbeit dokumentiert den Entwicklungs- und Implementierungsprozess, der im zugehörigen GitHub-Repository festgehalten wurde. Sowohl das Repository als auch Teile dieser Arbeit sind im Rahmen einer Zusammenarbeit mit einer parallel durchgeführten Bachelorarbeit entstanden, die eine vergleichbare Fragestellung behandelt. Da beide Arbeiten auf denselben theoretischen Grundlagen basieren, wurden diese gemeinsam erarbeitet. Auch der Vergleich der unterschiedlichen Implementierungsmethoden erfolgte in enger Abstimmung und wurde kollaborativ ausgearbeitet.

12. Filterimplementierung auf Mikrocontrollern

Die Zielhardware bei der Implementierung von digitalen IIR-Filtern auf Mikrocontroller ist das Lyrat V4.3 mit dem ESP32. Das Lyrat V4.3 ist ein Audio-Entwicklungsboard von Espressif Systems mit zwei Tensilica Xtensa LX6 Prozessorkernen welche bis zu 240 MHz Frequenz takten und einem 520 KB internem RAM. Dieses Board verfügt über einen ES8388 Audiocodec. Über diesen Codec werden Hardware-Ausstattung wie ein integriertes Stereo-Mikrofon für Audioaufnahme, AUX-IN und AUX-OUT Anschlüsse für externe Audioquellen und Audioausgabegeräte angesprochen. Über einen MicroSD-Kartenslot können WAV-Dateien als Audioquellen eingebunden werden. Diese Konfiguration ermöglicht sowohl die Verarbeitung von Audio-Streams in Echtzeit als auch die Filterung gespeicherter Audiodateien.

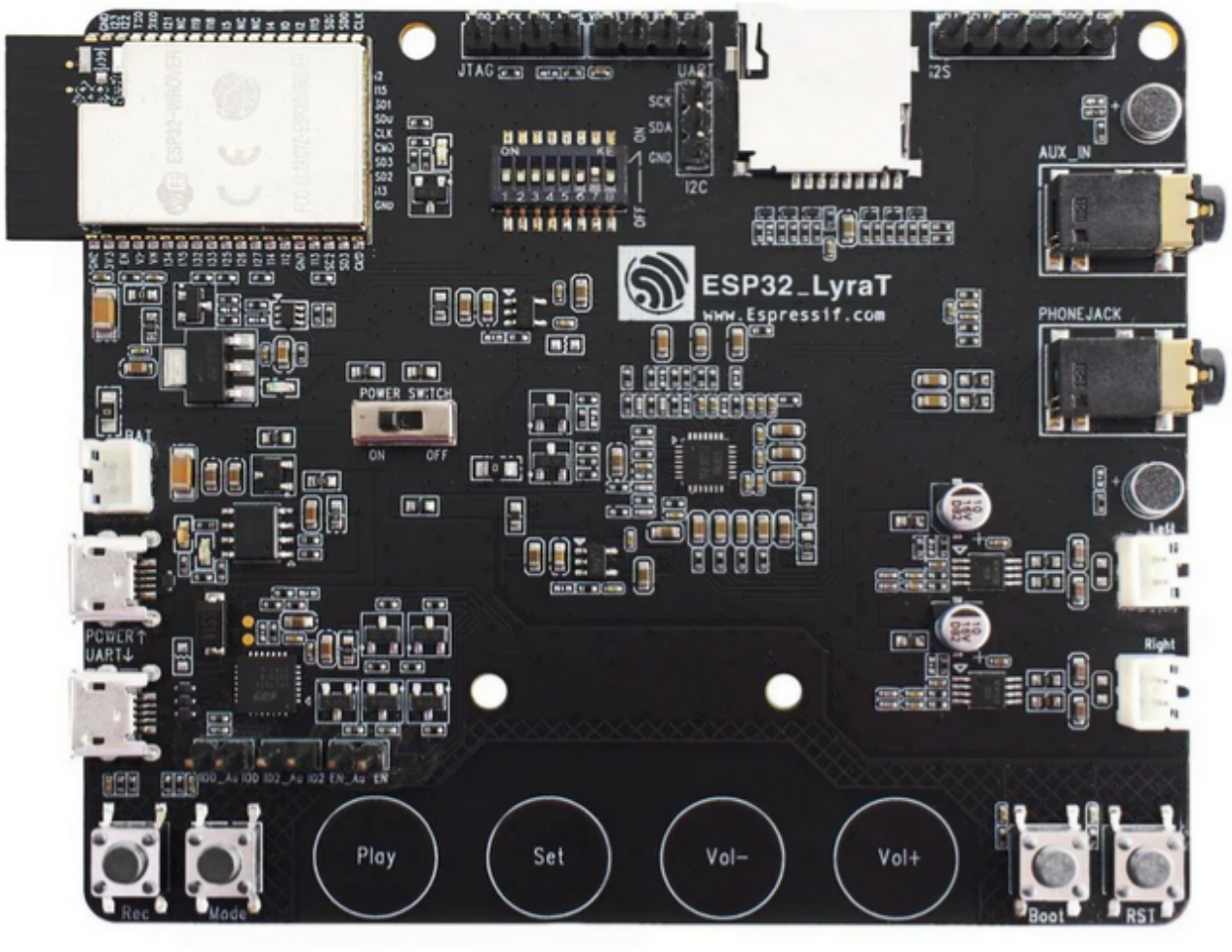


Abbildung 12.1.: ESP Lyrat V4.3

12.1. Arduino Implementierung

Das Arduino-Framework basiert auf C/C++ und bietet durch spezialisierte Audio-Bibliotheken wie die arduino-audio-tools sowie die arduino-audio-driver von Phil Schatzmann eine Basis für die Echtzeitaudioverarbeitung. Bei der Implementierung werden die Programme mittels der Arduino IDE verfasst, kompiliert und auf den ESP32 geflasht.

12.2. Micropython Implementierung

MicroPython verwendet eine interpretierte Python-Syntax, die eine schnellere Entwicklung und vereinfachtes Debugging ermöglicht, jedoch aufgrund geringerer Ausführungsgeschwindigkeit primär für Prototyping geeignet ist und für keine hohen Verarbeitungsprozesse geeignet ist. Für die

Implementierung in Micropython wird die ThonnyIDE mit dem Micropython Interpreter genutzt und der ESP32 wird mit einer Kompaktiblen Firmware geflasht.

12.3. Vorgehen bei der Implementierung

Für die Implementierung von digitalen Filtern auf Mikrocontrollern werde anhand der theoretischen Grundlagen die Strukturen durch ihre korrespondierenden Differenzengleichungen in Software umgesetzt. Diese Umsetzungen werden als Basis für weitere Verarbeitung von WAV-Datei genutzt. Im Folgenden werden die Strukturen in Arduino realisiert.

13. Implementierungserklärung der Biquad Filter in Arduino

Für die Implementierung der biquadratischen Filter in Arduino werden die Differenzengleichungen aus den Theoretischen Grundlagen entnommen und in Code umgesetzt. Die Filter sollen über das Einsetzten der `b` und `a` Koeffizienten implementiert werden.

13.1. Filter Basisklasse

```
class Filter
  {
  public:
    virtual ~Filter() {}
    virtual float filter(float) = 0;
  };
```

Diese abstrakte Basisklasse dient als Interface für alle Filterimplementierungsstrukturen. Der virtuelle Destruktor `virtual ~Filter() {}` stellt sicher, dass beim Löschen eines Objekts über einen Basisklassenzeiger der korrekte Destruktor der abgeleiteten Klasse aufgerufen wird. Die rein virtelle Methode `virtual float filter(float) = 0;` definiert die einheitliche Schnittstelle - jede abgeleitete Filterklasse muss eine `filter()` - Methode implementieren, die einen float-Wert entgegennimmt und einen gefilterten float-Wert zurückgibt. Diese Struktur ermöglicht die Verwendung der verschiedenen Filtertypen im Anwendiungsfall einer Kaskadierung.

13.2. Direktform 1

13.2.1. Private Member-Variablen

```
private:
  const float b_0;
  const float b_1;
  const float b_2;
  const float a_1;
  const float a_2;
```

```
  float x_1 = 0;
  float y_1 = 0;
  float y_2 = 0;
```

Die Filterkoeffizienten `b_0, b_1, b_2, a_1, a_2` sind als `const` deklariert, da sie nach der Initialisierung unveränderlich bleiben sollen. Dies entspricht der mathematischen Definition eines zeitinvarianten Systems. Die Verzögerungselemente `x_1, y_1, y_2` speichern die vorherigen Ein- und Ausgangswerte. Der Wert `a_0` wird nicht gespeichert, da durch die Normalisierung im Konstruktor alle Koeffizienten bereits durch `a_0` geteilt wurden.

13.2.2. Konstruktor

```
BiquadFilterDF1(const float (&b)[3], const float (&a)[3], float gain)
    : b_0(gain * b[0] / a[0]),
      b_1(gain * b[1] / a[0]),
      b_2(gain * b[2] / a[0]),
      a_1(a[1] / a[0]),
      a_2(a[2] / a[0])
{
}
```

Der Konstruktor verwendet Initialisierungslisten für optimale Performance. Die Koeffizienten werden direkt bei der Objekterstellung normalisiert, indem alle durch `a[0]` geteilt werden. Dies entspricht der Standardform der Differenzengleichung, wo der führende Koeffizient des Nenners auf 1 normiert wird. Der `gain` Parameter wird in die Zählerkoeffizienten eingerechnet, was mathematisch äquivalent zur Multiplikation der gesamten Übertragungsfunktion mit dem Verstärkungsfaktor ist. Die Array-Referenz-Syntax `const float (&b)[3]` stellt sicher, dass exakt drei Elemente übergeben werden müssen.

13.2.3. Filter Methode

```
float filter(float x_0)
{
  float x_2 = x_1;
  x_1 = x_0;

  float y_0 = b_0 * x_0 + b_1 * x_1 + b_2 * x_2 - a_1 * y_1 - a_2 * y_2;

  y_2 = y_1;
  y_1 = y_0;

  return y_0;
}
```

Diese Implementierung wird durch die Struktur der DF1 umgesetzt. Zuerst werden die Eingangsverzögerungen aktualiseirt: `x_2` erhält den vorherigen Wert von `x_1`, dann wird `x_1` auf den aktuellen Eingangswert `x_0` gesetzt. Die Differenzengleichung wird direkt implementiert: `y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] - a_1 * y[n-1] - a_2 * y[n-2]`. Im Anschluss werden die Ausgangsverzögerungen für die nächste Iteration aktualisiert. Diese Reihenfolge ist kritisch, da die Berechnung vor Aktualisierung der Ausgangsverzögerungen erfolgen muss.

13.3. Direktform 2

13.3.1. Private Member Variablen

```
private:
  const float b_0;
  const float b_1;
  const float b_2;
  const float a_1;
  const float a_2;

  float w_0 = 0;
  float w_1 = 0;
```

Die DF2-Struktur benötigt nur zwei Verzögerungselemente `w_0, w_1` anstatt der vier in DF1. Dies reduziert den Speicherbedarf um die 50%. Die `w` Variablen repräsentieren die internen Knotenpunkte der DF2-Struktur, wo sowohl die Rückkopplung als auch die Vorwärtskopplung zusammenlaufen.

13.3.2. Konstruktor

```
BiquadFilterDF2(const float (&b)[3], const float (&a)[3], float gain)
    : b_0(gain * b[0] / a[0]),
      b_1(gain * b[1] / a[0]),
      b_2(gain * b[2] / a[0]),
      a_1(a[1] / a[0]),
      a_2(a[2] / a[0])
{
}
```

Der Konstruktor ist identisch zur DF1 Implementierung, da die Koeffizientennormalisierung unabhängig von der internen Filterstruktur ist. Die mathematische Übertragungsfunktion bleibt dieselbe, nur die interne Realisierung unterscheidet sich

13.3.3. Filter Methode

```
float filter(float x_0)
{
  float w_2 = w_1;
  w_1 = w_0;
  w_0 = x_0 - a_1 * w_1 - a_2 * w_2;

  float y_0 = b_0 * w_0 + b_1 * w_1 + b_2 * w_2;

  return y_0;
}
```

Die DF2 Implementierung teilt die Berechnung in zwei Phasen: Zuerst wird der interne Zustand `w_0` berechnet, der das Eingangssignal minus der Rückkopplung darstellt. Dies entspricht der Implementierung des Nennerpols der Übertragungsfunktion. Dann wird der Ausgang durch die Vorwärtskopplung mit den `b` Koeffizienten berechnet, was dem Zählerpolynom entspricht. Die Verzögerungselemente werden vor der `w_0` Berechnung aktualisiert, damit die korrekten vorherigen Werte verwendet werden.

13.4. Transponierte Direktfrom 2

13.4.1. Private Member Variablen

```
private:
  const float b_0;
  const float b_1;
  const float b_2;
  const float a_1;
  const float a_2;

  float s_1 = 0;
  float s_2 = 0;
```

Die TDF2 Struktur verwendet Zustandsvariablen `s_1, s_2`, die als "shift register" fungieren. Diese Struktur ist die transponierte Version der DF2, was bedeutet, dass der Signalfluss umgekehrt wird: die Ein- und Ausgänge werden vertauscht, sowohl als auch die Richtung der Verzögerungselemente wird umgekehrt.

13.4.2. Konstruktor

```
BiquadFilterTDF2(const float (&b)[3], const float (&a)[3], float gain)
    : b_0(gain * b[0] / a[0]),
```

```
      b_1(gain * b[1] / a[0]),
      b_2(gain * b[2] / a[0]),
      a_1(a[1] / a[0]),
      a_2(a[2] / a[0])
{
}
```

Auch hier ist der Konstruktor identisch zu den anderen Implementierungen, da die Koeffizientennormalisierung eine mathematische Anforderung ist, die unabhängig von der gewählten Realisierungsform gilt.

13.4.3. Filter Methode

```
float filter(float x_0)
{
  float y_0 = s_1 + b_0 * x_0;

  s_1 = s_2 + b_1 * x_0 - a_1 * y_0;
  s_2 = b_2 * x_0 - a_2 * y_0;

  return y_0;
}
```

Bei der TDF2 Implementierung setzt sich der Ausgangswert aus der addition aus dem ersten Zustandsregister `s_1` mit dem verstärkten Eingangswert. Die Zustandsregister werden dann für die nächste Iteration aktualisiert. `s_1` wird zum nächsten Wert von `s_2` addiert mit den gewichteten Ein- und Ausgangswerten. `s_2` wird komplett neu berechnet. Diese Struktur hat den Vorteil, dass der Ausgangswert sehr früh im Berechnungszyklus verfügbar ist, was bei Pipeline Implementierungen sich als vorteilhaft erweisen kann.

13.5. Eingabe Koeffiziente

```
const float b_0 = f;
const float b_1 = f;
const float b_2 = f;
const float a_0 = f;
const float a_1 = f;
const float a_2 = f;
```

Bei den Eingabe-Koeffezienten wird der `f` Suffix an die float Literalen angehängt, sodass sicher gestellt wird das die Werte als singel-precision floating-point interpretiert werden, was bei Arduino-System von hoher Wichtigkeit ist.

13.6. Koeffizienten Arrays und Filter Instanziierung

```
const float b_coefficients[] = { b_0, b_1, b_2};
const float a_coefficients[] = { a_0, a_1, a_2};
const float gain = 1;

BiquadFilterTDF2 biquad(b_coefficients, a_coefficients, gain);
```

Die Koeffizienten werden in separaten Arrays gespeichert und an den Konstruktor übergeben. Dies ermöglicht eine saubere Trennung zwischen Filterdesign (Koeffizienten) und Implementierung (Filterklasse). Über den `gain` Paramter kann auf den zu Implementierenden Filter eine gewünschte Verstärkung angewandt werden. Bei einem Wert von 1 wird keine zusätzliche Verstärkung angewendet. Bei dieser Filter Instanziierung wird die TDF2 verwendet mir ihrem korosponierenden Klassennamen. Somit wird das Objekt `biquad` durch die Klasse `BiquadFilterTDF2` instanziiert. Innerhalb des Objekts werden die Koeffizienten mit dem Verstärkungsfaktor an die jeweilige Filterklasse übergeben. Der Implementierte Filter steht bereit zur Verwendung.

13.7. Filter Anwendung

```
float filtered = biquad.filter(Input);
```

Die direkte Anwendung erfolgt durch den Aufruf der `filter()`-Methode auf dem Filterobjekt `biquad`. Der Eingangswert `Input` wird der Methode übergeben, durch die spezifische Filterimplementierung entsprechend der definierten Differenzengleichung verarbeitet und als transformierter Ausgangswert filtered zurückgegeben.

14. Filterung einer WAV Datei in Arduino

Zur Demonstration von digitalen Filtern auf Mikrocontrollern wird eine WAV-Datei mittels eines ESP32 gefiltert. Die Audio-Datei wird über eine SD-Karte eingelesen, gefiltert und als gefilterte Variante abgespeichert.

Zu beachten ist, dass nur 16-Bit-WAV-Dateien unterstützt werden. Die Datei kann jedoch Mono oder Stereo mit und einer beliebigen Samplerate sein. Die WAV-Datei, die gefiltert werden soll, muss zu `original.wav` umbenannt werden bevor diese auf die SD-Karte übertragen wird.

14.1. Vorbereitung der WAV-Filterung

```
#include "SD_MMC.h" // SD-Karten-Support für ESP32 mit SD_MMC-Anschluss
```

Durch die Einbindung vom SD_MMC.h wird eine erleichterte Nutzung von SD-Karten mit einem ESP32 geboten, als auch die Möglichkeit für 1-Wire-Modus sowie 4-Wire-Modus.

```
// --- WAV Hilfsfunktionen ---
uint32_t readLEUint32(uint8_t* buf) {
  return (uint32_t)buf[0] | ((uint32_t)buf[1] << 8)
  | ((uint32_t)buf[2] << 16) | ((uint32_t)buf[3] << 24);
}
uint16_t readLEUint16(uint8_t* buf) {
  return (uint16_t)buf[0] | ((uint16_t)buf[1] << 8);
}
```

Diese Hilfsfunktionen konvertieren Byte-Arrays aus WAV-Dateien in Ganzzahlen im Little-Edian-Format um. `readLEUint32()` kombiniert 4 Bytes zu einem 32-Bit-wert, `readLEUint16()` kombiniert 2 Bytes zu einem 16-Bit-Wert, wobei das niedrigste Byte zuerst kommt.

```
// WAV-Header für die Ausgabedatei schreiben
void writeWavHeader(File &file, uint32_t dataSize, uint16_t channels,
uint32_t sampleRate, uint16_t bitsPerSample) {
  const char* riff = "RIFF";
  const char* wave = "WAVE";
  const char* fmt  = "fmt ";
  const char* data = "data";

  uint32_t chunkSize = 36 + dataSize;
```

```
  uint16_t blockAlign = channels * bitsPerSample / 8;
  uint32_t byteRate = sampleRate * blockAlign;
  uint32_t subchunk1Size = 16;
  uint16_t audioFormat = 1; // PCM

  file.seek(0);
  file.write((const uint8_t*)riff, 4);
  file.write((uint8_t*)&chunkSize, 4);
  file.write((const uint8_t*)wave, 4);
  file.write((const uint8_t*)fmt, 4);
  file.write((uint8_t*)&subchunk1Size, 4);
  file.write((uint8_t*)&audioFormat, 2);
  file.write((uint8_t*)&channels, 2);
  file.write((uint8_t*)&sampleRate, 4);
  file.write((uint8_t*)&byteRate, 4);
  file.write((uint8_t*)&blockAlign, 2);
  file.write((uint8_t*)&bitsPerSample, 2);
  file.write((const uint8_t*)data, 4);
  file.write((uint8_t*)&dataSize, 4);
}
```

Die `writeWavHeader`-Funktion schreibt einen WAV-Header in die Ausgabedatei. Die Funktion erstellt die erforderlichen Chunks (RIFF, WAVE, fmt, data) mit den übergebenen Audio-Parametern und berechnet automatisch die abgeleiteten Werte wie Byte-Rate und Block-Alignment. Der Header wird im Little-Endian-Format geschrieben.

```
// --- Variablen für die Ein- und Ausgangsdatei --
File wavFile;       // Originale Eingangsdatei
File filteredFile;  // Gefiltere Ausgangsdatei
```

Hilfsvariablen für die originale Eingangsdatei als auch die gefilterte Ausgangsdatei.

Der Gesamtprozess bestehend aus einlesen, filtern und neuen WAV-Datei erstellen wird innerhalb der `Setup()`-Funktion durchgeführt.

```
void setup() {
  Serial.begin(115200);
```

Zunächst wird die serielle Kommunikation mit 115200 Baud initialisiert, um Informationen des Mikrocontroller in der Konsole auslesen zu können.

```
// SD-Karte im 4-Wire Modus initialisieren
if (!SD_MMC.begin("/sdcard", false)) {  //false = 4-Wire, true - 1-Wire
  Serial.println("Fehler beim Initialisieren der SD-Karte.");
  return;
}
Serial.println("SD-Karte gefunden.");
```

Die SD-Karte wird im 4-Wire-Modus initialisiert, welcher eine höhere Datenübertragungsrate ermöglicht als der 1-Wire-Modus. Durch den Parameter `false` wird der 4-Wire-Modus aktiviert, während `true` den 1-Wire-Modus aktiviert. Wird keine SD-Karte gefunden oder besteht ein Problem bei der Initialisierung, wird eine Fehlermeldung ausgegeben und das Programm beendet.

```
// Eingangs WAV Datei öffnen
wavFile = SD_MMC.open("/original.wav", FILE_READ);
if (!wavFile) {
  Serial.println("Fehler beim Öffnen der WAV-Datei.");
  return;
}

// WAV Header auslesen
if (!readWavHeader()) {
  wavFile.close();
  return;
}
```

Die originale WAV-Datei wird im Lesemodus geöffnet. Für den Fall, dass das Öffnen fehlschlägt, wird dementsprechende Fehlermeldung ausgegeben und das Programm beendet. Bei erfolgreichen Öffnen wird die Funktion `readWavHeader` aufgerufen und der Header der Eingangsdatei analysiert. Falls diese Funktion `false` zurückgibt, wird die Datei geschlossen und das Programm beendet.

```
bool readWavHeader() {
  uint8_t riffHeader[12];
  if (wavFile.read(riffHeader, 12) != 12) {
    Serial.println("Fehler beim Lesen des RIFF-Headers.");
    return false;
  }
  //Prüfen auf "RIFF" und "WAVE"
  if (memcmp(riffHeader, "RIFF", 4) != 0 || memcmp(&riffHeader[8],
   "WAVE", 4) != 0) {
    Serial.println("Keine gültige WAV-Datei.");
    return false;
  }
```

Die Header-Analyse-Funkion beginnt mit der Validierung der WAV-Datei Signatur. Dazu werden die ersten 12 Bytes der Datei gelesen, die den RIFF-Header bilden. Anschließend wird überprüft, ob die Bytes 0-3 die Zeichenkette “RIFF” und die Bytes 8-11 die Zeichenkette “WAVE” enthalten. Diese Signaturen sind für alle gültigen WAV-Dateien notwendig und identifizieren das genaue Dateiformat.

```
// Variablen für WAV-Informationen
uint16_t audioFormat = 0, numChannels = 0, bitsPerSample = 0;
uint32_t sampleRate = 0, dataSize = 0;
bool fmtFound = false, dataFound = false;
```

Für die Chunk-Analyse werden die folgenden Variablen initialisiert, welche die Audio-Parameter speichern. Die 16-Bit Variablen `audioFormat`, `numChannels` und `bitsPerSample` enthalten die grundlegenden Format-Informationen, während `sampleRate` und `dataSize` als 32-Bit-Werte die Abtastrate und Datengröße speichern. Die booleschen Flags `fmtFound` und `dataFound` verfolgen, die notwendigen Chunks bereits gefunden wurden.

```
// Suche nach "fmt " und "data" Chunks
while (wavFile.position() < wavFile.size()) {
  uint8_t chunkHeader[8];
  if (wavFile.read(chunkHeader, 8) != 8) {
    Serial.println("Fehler beim Lesen eines Chunk-Headers.");
    break;
  }
  uint32_t chunkSize = readLEUint32(&chunkHeader[4]);
```

Der Chunk-Parser durchläuft systematisch die WAV-Datei, um relevante Datenabschnitte zu finden. Die WAV-Datei ist in Chunks unterteilt, die jeweils mit einem 8-Byte-Header beginnt: 4 Bytes für die Chunk-ID und 4 Bytes für die Größe im Little-Endian-Format. Die Schleife läuft bis zum Dateiende oder bis ein Lesefehler auftritt.

```
//Format-Chunk gefunden
if (memcmp(chunkHeader, "fmt ", 4) == 0) {
  uint8_t fmtChunk[chunkSize];
  if (wavFile.read(fmtChunk, chunkSize) != chunkSize) {
    Serial.println("Fehler beim Lesen des fmt-Chunks.");
    break;
  }
  audioFormat = readLEUint16(&fmtChunk[0]);
  numChannels = readLEUint16(&fmtChunk[2]);
  sampleRate = readLEUint32(&fmtChunk[4]);
  bitsPerSample = readLEUint16(&fmtChunk[14]);
  fmtFound = true;

  // Parameter der Eingangs-WAV
  Serial.printf("AudioFormat: %d\n", audioFormat);
  Serial.printf("Kanäle: %d\n", numChannels);
  Serial.printf("SampleRate: %d\n", sampleRate);
  Serial.printf("Bits pro Sample: %d\n", bitsPerSample);

  if (chunkSize % 2 == 1) wavFile.seek(wavFile.position() + 1);
}
```

Wenn der Chunk-Header “fmt” gefunden wird, werden die Chunk-Daten in einen Puffer geladen und die wichtigsten Audio-Parameter extrahiert: Audi-Format(PCM), Kanalzahl(Mono oder Stereo), Samplerate und die Bits pro Sample aus festen Byte-Positionen. Die Parameter werden zur Kontrolle ausgegeben und bei ungerader Chunk-Größe wird ein Padding-Byte übersprungen.

```
// Daten-Chunk gefunden
else if (memcmp(chunkHeader, "data", 4) == 0) {
  dataSize = chunkSize;
  dataFound = true;
  break;
}
```

Der Data-Chunk enthält die eigentlichen Audio-Daten mit den Samples. Sobald dieser gefunden wird, wird seine Größe gespeichert und die Suche endet. Der Dateizeiger steht dann am Beginn der Audio-Samples für die weitere Verarbeitung.

```
  // Irrelavante Chucks überspringen
  else {
    wavFile.seek(wavFile.position() + chunkSize);
    if (chunkSize % 2 == 1) wavFile.seek(wavFile.position() + 1);
  }
}
```

Irrelevante Chunks, wie LIST oder INFO, werden übersprungen, indem der Dateizeiger um die Chunk-Größe voranbewegt wird. Bei ungerader Chunk-Größe wird ein zusätzliches Padding-Byte übersprungen.

```
// Überprüfen ob alle nötigen Informationen gefunden wurden
if (!fmtFound || !dataFound) {
  Serial.println("WAV-Header unvollständig oder Datenchunk nicht gefunden.");
  return false;
}

if (audioFormat != 1) {
  Serial.println("Nur PCM WAV-Dateien werden unterstützt.");
  return false;
}

if (bitsPerSample != 16) {
  Serial.println("Nur 16-Bit PCM wird unterstützt.");
  return false;
}

if (numChannels != 1 && numChannels != 2) {
  Serial.println("Nur Mono oder Stereo wird unterstützt.");
  return false;
}
```

Es wird überprüft, ob Format- und Data-Chunk gefunden wurden und ob die Datei das kompatible Format besitzt. Entspricht die Datei nicht dem vorgesetzten Format wird eine Fehlermeldung ausgeben und das Programm beendet.

```
// Ausgabedatei erstellen und Header schreiben
filteredFile = SD_MMC.open("/gefiltert.wav", FILE_WRITE);
if (!filteredFile) {
  Serial.println("Fehler beim Öffnen der Ausgabedatei.");
  return false;
}

writeWavHeader(filteredFile, dataSize, numChannels, sampleRate, bitsPerSample);

Serial.printf("Gesamtanzahl Frames (Samples pro Kanal): %d\n",
              dataSize / (numChannels * bitsPerSample / 8));
Serial.println("Starte Filterung...");

// Filterprozess starten
filterAudio(numChannels, dataSize, bitsPerSample);

Serial.println("Filterung abgeschlossen.");

wavFile.close();
filteredFile.close();

return true;
```

Die Ausgabedatei “gefiltert.wav” wird im Schreibmodus erstellt und erhält einen neuen WAV-Header mit den identischen Parametern der Eingangsdatei. Würde man keinen neuen WAV-Header erstellen, sondern den bestehenden verwenden, würde die resultierende Datei fehlerhaft enden. Es werden die Anzahl der Frames berechnet und ausgeben, und im Anschluss wird die Sample-für-Sample Filterung über die `filterAudio()`-Funktion gestartet. Nach der Filterung werden beide Dateien geschlossen und endet damit das Programm.

14.2. Filterprozess der WAV-Datei

```
// --- Filterprozess auf Samples der WAV anwenden ---
void filterAudio(uint16_t numChannels, uint32_t dataSize, uint16_t bitsPerSample) {
  uint16_t blockAlign = numChannels * bitsPerSample / 8;
  const int bufferFrames = 512;
  int16_t buffer[bufferFrames * numChannels]; // Zwischenspeicher für Samples

  uint32_t totalFrames = dataSize / blockAlign;
  uint32_t framesLeft = totalFrames;

  while (framesLeft > 0) {
    int framesToRead = (framesLeft > bufferFrames) ? bufferFrames : framesLeft;
    int bytesToRead = framesToRead * blockAlign;
```

```
// Rohdaten lesen
int bytesRead = wavFile.read((uint8_t*)buffer, bytesToRead);
if (bytesRead != bytesToRead) {
  Serial.println("Fehler beim Lesen der Samples.");
  break;
}

int samplesInBuffer = bytesRead / 2; // 2 Bytes = 16 Bit pro Sample


// Samples einzeln filtern
for (int i = 0; i < samplesInBuffer; i += numChannels) {
  if (numChannels == 1) {
    float filteredSample = filterL.filter((float)buffer[i]);
    buffer[i] = constrain((int)filteredSample, -32768, 32767);
  }
  else {
    float filteredL = filterL.filter((float)buffer[i]);
    float filteredR = filterR.filter((float)buffer[i + 1]);
    buffer[i] = constrain((int)filteredL, -32768, 32767);
    buffer[i + 1] = constrain((int)filteredR, -32768, 32767);
  }
}
// Gefilterte Daten in neue Datei schreiben
filteredFile.write((uint8_t*)buffer, bytesRead);
framesLeft -= framesToRead;
```

Die `filterAudio`-Funktion führt die Sample-für-Sample-Verabreitung der Audio-Daten durch.

Blockweise Verarbeitung: Die Audio-Daten werden in 512-Frame-Blöcken eingelesen, anstatt die gesamte Datei in den Speicher zu laden. Die ermöglicht auch die Verarbeitung großer Dateien mit begrenztem Arbeitsspeicher.

Filteranwendung: Jedes der Sample wird durch die digitalen Filter gesendet. Bei Mono wird jeweils jedes Sample durch eine einzige Filterinstanz (`filterL`) verarbeitet. Bei Stereo liegen die 16-Bit-Samples im Interleaved-Format vor, bei welchem sich der linken und rechten Kanalsamples abwechseln(LRLRLR…). Die Schleife springt um `numChannels` Positionen (`i +=numChannels`), wodurch `buffer[i]` stets den linken und `buffer[i]` den rechten Kanal adressiert. Beide Kanäle werden durch separate Filterinstanzen (`filterL` für links, `filterR` für rechts) unabhängig voneinander gefiltert, was die Stereo-Trennung gewährleistet.

Wertebreichsschutz: Da digitale Filter die Amplitude verstärken können, werden die gefilterten Werte mit `constrain()` auf den gültigen 16-Bit-Bereich begrenzt. Dies verhindert Übersteuerung und damit verbundene Verzerrungen.

Kontinuierliche Ausgabe: Die verarbeiteten Samples werden sofort in die Ausgabedatei geschrieben, wodurch ein kontinuierlicher Datenfluss ohne große Zwischenspeicherung gewährleistet wird.

14.3. Filterkaskadierung

Die Kaskadierung wird durchgeführt, indem die Filterung durch mehrere Stufen durchgeführt wird. Diese Stufen können durch Biquad-Filter oder Segmente einer SOS-Matrix definiert werden.

```
#define NUM_STAGES 2 // Anzahl der Filterstufen

// Erste Sektion
const float gain_s1 = 1.0;
const float b_coefficients_s1[] = { b_0_s1, b_1_s1, b_2_s1};
const float a_coefficients_s1[] = { a_0_s1, a_1_s1, a_2_s1};
// Zweite Sektion
const float gain_s2 = 1.0;
const float b_coefficients_s2[] = { b_0_s2, b_1_s2, b_2_s2};
const float a_coefficients_s2[] = { a_0_s2, a_1_s2, a_2_s2};

BiquadFilterTDF2 filterL[NUM_STAGES] = {
  BiquadFilterTDF2(b_coefficients_s1, a_coefficients_s1, gain_s1),
  BiquadFilterTDF2(b_coefficients_s2, a_coefficients_s2, gain_s2),
};

BiquadFilterTDF2 filterR[NUM_STAGES] = {
  BiquadFilterTDF2(b_coefficients_s1, a_coefficients_s1, gain_s1),
  BiquadFilterTDF2(b_coefficients_s2, a_coefficients_s2, gain_s2),
};
```

Anhand des Parameters `NUM_STAGES` kann die Anzahl der Biquad-Sektionen definiert werden. Jede Sektion hat ihre eigenen `b`- und `a`-Koeffizienten sowie einen `gain`-Parameter mit den entsprechenden Bezeichnungen. Die Filterinstanzierung erfolgt durch Arrays von Biquad-Filtern, wobei jedes Array-Elemement eine separate Filtersstufe mit eigenen Koeffizienten repräsentiert.

Werden identische Filterparameter in mehrere Filterstufen implementiert, wird die Filtercharakteristik der vorherigen Stufe verstärkt. Alternativ können Filter höherer Ordnung implementiert werden, indem die SOS-Matrix etappenweise auf die einzelnen Filterstufen aufgeteilt wird.

```
// Samples einzeln filtern - KASKADIERT
for (int i = 0; i < samplesInBuffer; i += numChannels) {
  if (numChannels == 1) {
    float sample = (float)buffer[i];
    for (int s = 0; s < NUM_STAGES; s++) {
      sample = filterL[s].filter(sample);
    }
```

```
    buffer[i] = constrain((int)sample, -32768, 32767);
  }
  else {
    float left = (float)buffer[i];
    float right = (float)buffer[i + 1];
    for (int s = 0; s < NUM_STAGES; s++) {
      left = filterL[s].filter(left);
      right = filterR[s].filter(right);
    }
  buffer[i] = constrain((int)left, -32768, 32767);
  buffer[i + 1] = constrain((int)right, -32768, 32767);
}
```

Bei der kaskadierten Filterung werden die Filterstufen in Serie Implementiert, bei welcher die Samples durch jede Stufe sequenziell verarbeitet werden. Das Ausgangssignal einer Stufe wird zum Eingangssignal der nächsten Stufe, wodurch eine erhöhte Filterordnung oder verstärkte Filterwirkung ermöglicht. Bei Mono druchläuft jedes Sample alle `NUM_STAGES` Filterstufen des `filterL`-Arrays, bei Stereo werden beide Kanäle parallel durch ihre jeweiligen Filterkaskaden(`filterL` und `filterR`) verarbeitet. Nach der kompletten Kaskadierung werden die gefilterten Daten auf den 16-Bit-Bereich begrenzt und zurückgeschrieben.

15. Implementierungserklärung der Biquad Filter in Micropython

Für die Implementierung der biquadratischen Filter in Micropython werden die Differenzengleichungen aus den theoretischen Grundlagen entnommen und in Code umgesetzt. Die Filter werden über das Einsetzen der `b` und `a` Koeffizienten implementiert. Die DF1, DF2 und TDF2 werden als Klassen umgesetzt.

15.1. Micropython-Native Optimierung

```
import micropython

@micropthon.native
def filter(self,x0):
```

Der `@micropython.native` Decorator kompiliert die `filter()` Methoden zu nativem Maschinencode, was eine erhebliche Leistungssteigerung bei rechenintensiven Operationen ermöglicht.

15.2. `__slots__`Optimierung

```
__slots__=('b0', 'b1', 'b2', 'a2', 'gain')`
```

Alle Filterklassen verwenden `__slots__`, um den Speicherverbrauch zu reduzieren und die Attributzugriffe zu beschleunigen. Dies verhindert, dass Python ein dynamisches Dictionary für jede Instanz erstellt, wodurch sowohl Speicher als auch Zugriffszeit gespart werden.

15.3. Direktform 1

15.3.1. Instanzvariablen

```
class BiquadFilterDF1:
    __slots__ = ('b0', 'b1', 'b2', 'a1', 'a2', 'gain', 'x1', 'x2', 'y1', 'y2')
```

Die Filterkoeffizienten `b0, b1, b2, a1, a2` werden nach der Initialisierung im Konstruktor nicht mehr verändert, was der mathematischen Definition eines zeitinvarianten Systems entspricht. Die Verzögerungselemente `x1, x2` speichern die vorherigen Eingangswerte, während `y1, y2` die vorherigen Ausgangswerte speichern. Der Wert `a0` wird nicht gespeichert, da durch die Normalisierung im Konstruktor alle Koeffizienten bereits durch `a[0]` geteilt wurden.

15.3.2. Konstruktor

```
def __init__(self, b, a, gain=1):
    self.gain = gain
    self.b0 = self.gain * (b[0] / a[0])
    self.b1 = self.gain * (b[1] / a[0])
    self.b2 = self.gain * (b[2] / a[0])
    self.a1 = a[1] / a[0]
    self.a2 = a[2] / a[0]
    self.x1 = 0.0
    self.x2 = 0.0
    self.y1 = 0.0
    self.y2 = 0.0
```

Der Konstruktor normalisiert alle Koeffizienten druch die Division mit `a[0]`, wodurch die Standardform der Differenzengleichung erreicht wird. Dies entspricht der Standardform, wo der führende Koeffizient des Nenners auf 1 normiert wird. Der `gain` Parameter wird in die Zählerkoeffizienten eingerechnet, was mathematisch äquivalent zur Multiplikation der gesamten Übertragungsfunktion mit dem Verstärkungsfaktor ist. Die Verzögerungselemente werden auf `0.0` initialisiert, was einem System ohne vorherigen Werte entspricht.

15.3.3. Filter Methode

```
@micropython.native
def filter(self, x0):
    y0 = (self.b0 * x0 + self.b1 * self.x1 + self.b2 * self.x2
        - self.a1 * self.y1 - self.a2 * self.y2)
    self.x2 = self.x1
    self.x1 = x0
    self.y2 = self.y1
    self.y1 = y0
    return y0
```

Für die Implementierung der DF1 wird die Differenzengleichung dieser Struktur direkt implementiert: `y[n] = b0 * x[n] + b1 * x[n-1] + b2 * [x-2] - a1 * y[n-1] - a2 * y[n-2]`. Anschließend werden die Verzögerungselemente für die nächste Iteration aktualisiert. Die Eingangsverzögerung werden durch `self.x2 = self.x1` und `self.x1 = x0` verschoben, während die Ausgangsverzögerung

durch `self.y2=self.y1` und `self.y1 = y0` aktualisiert werden. Diese Reihenfolge ist kritisch, da die Berechnung vor der Aktualisierung der Ausgangswerte erfolgen muss.

15.4. Direktform 2

15.4.1. Instanzvariablen

```
class BiquadFilterDF2:
    __slots__ = ('b0', 'b1', 'b2', 'a1', 'a2', 'gain', 'w0', 'w1', 'w2')
```

Die DF2-Struktur benötigt drei Verzögerungselemente `w0, w1, w2` statt der vier in der DF1. Dies reduziert den Speicherbedarf gegenüber der DF1. Die `w` Variablen repräsentieren die internen Knotenpunkte der DF2-Struktur, wo sowohl die Rückkopplung als auch die Vorwärtskopplung zusammenlaufen.

15.4.2. Konstruktor

```
    def __init__(self, b, a, gain=1):
        slef.gain = gain
        self.b0 = self.gain * (b[0] / a[0])
        self.b1 = self.gain * (b[1] / a[0])
        self.b2 = self.gain * (b[2] / a[0])
        self.a1 = a[1] / a[0]
        self.a2 = a[2] / a[0]
        self.w0 = 0.0
        self.w1 = 0.0
        self.w2 = 0.0
```

Der Konstruktor ist geradezu identisch zur DF1 Implementierung, da die Koeffizientennormalisierung unabhängig von der internen Filterstruktur ist. Der Unterschied liegt in der Vordefinition der Verzögerungselemente `self.w0, self.w1, self.w2`, welchen der Wert von 0.0 zugewiesen wird.

15.4.3. Filter Methode

```
    @micropython.native
    def filter(self, x0):
        y0 = self.b0 * self.w0 + self.b1 * self.w1 + self.b2 * self.w2
        self.w0 = x0 - self.a1 * self.w1 - self.a2 * self.w2
        self.w2 = self.w1
        self.w1 = self.w0
        return y0
```

Die DF2 Implementierung teilt die Berechnung in zwei Phasen: Zuerst wird der Ausgang aus den aktuellen Zustandsvariablen und den Zählerkoeffizienten berechnet. Dies entspricht der Implementierung des Zählerpolynoms der Übertrangungsfunktion. Dann wird der neue Zustand `self.w0` berechnet, der das Eingangssignal minus der Rückkopplung darstellt. Die Verzögerungselemente werden durch `self.w2 = self.w1` und `self.w1 = self.w0` für die nächste Iteration verschoben.

15.5. Transponierte Direktform 2

15.5.1. Instanzvariablen

```
class BiquadFilterTDF2:
    __slots__ = ('b0', 'b1', 'b2', 'a1', 'a2', 'gain', 's1', 's2')
```

Die TDF2-Struktur verwendet Zustandsvariablen `s1, s2`, die als "shift register" fungieren. Diese Struktur ist die transponierte Version der DF2, was bedeutet, dass der Signalfluss umgekehrt wird: die Ein- und Ausgänge werden vertauscht, sowie die Richtung der Verzögerungselemente wird umgekehrt.

15.5.2. Konstruktor

```
  def __init__(self, b, a, gain=1):
      slef.gain = gain
      self.b0 = self.gain * (b[0] / a[0])
      self.b1 = self.gain * (b[1] / a[0])
      self.b2 = self.gain * (b[2] / a[0])
      self.a1 = a[1] / a[0]
      self.a2 = a[2] / a[0]
      self.s1 = 0.0
      self.s2 = 0.0
```

Auch hier ist der Konstruktor identisch zu den anderen Implementierungen, da die Koeffizientennormalisierung eine mathematische Anforderung ist, die unabhängig von der gewählten Realisierungsform gilt. In diesem Konstruktor werden die Variablen `self.s1, self.s2` auf 0.0 gesetzt.

15.5.3. Filter Methode

```
    @micropython.native
    def filter(self, x0):
        y0 = self.b0 * x0 + self.s1
        self.s1 = self.s2 + self.b1 * x0 - self.a1 * y0
        self.s2 = self.b2 * x0 - self.a2 * y0
        return y0
```

Bei der TDF2 Implementierung setzt sich der Ausgangswert aus der Addition aus dem ersten Zustandsregister `self.s1` mit dem verstärkten Eingangswert zusammen. Die Zustandsregister werden dann für die nächste Iteration aktualisiert. `self.s1`wird zum nächsten Wert von `self.s2` addiert mit den gewichteten Ein- und Ausgangswerten. `self.s2` wird komplett neu berechnet. Diese Struktur hat den Vorteil, dass der Ausgangswert sehr früh im Berechnungszyklus verfügbar ist, was bei Pipeline Implementierung vorteilhaft ist.

15.6. Filter Anwendung

```
from biquad import BiquadFilterTDF2
```

Für die Verwendung der Biquad-Filter wird aus der biquad.py die gewünschte Filterimplementierung importiert.

```
b = [1.0, 0.5, 0.25]
a = [1.0, -0.3, 0.1]
gain = 1.0

biquad = BiquadFilterTDF2(b, a, gain)
```

Die Koeffizienten werden in separaten Listen gespeichert und an den Konstruktor übergeben. Über den `gain` Parameter kann auf den zu implementierenden Filter eine gewünschte Verstärkung angewandt werden. Bei dieser Filter Instanziierung wird die TDF2 verwendet mit ihren entsprechenden Klassennamen. Somit wird das Objekt `biquad` durch die Klasse `BiquadFilterTDF2` instanziiert. Innerhalb des Objekts werden die Koeffizienten mit dem Verstärkungsfaktor an die jeweilige Filterklasse übergeben und der Filter steht bereit zur Verwendung.

```
filtered = biquad.filter(input)
```

Die direkte Anwendung erfolgt durch den Aufruf der filter()-Methode auf dem Filterobjekt `biquad`. Der Eingangswert `input` wird der Methode übergeben, durch die spezifische Filterimplementierung entsprechend der definierten Differenzengleichung verarbeitet und als transformierter Ausgangswert `filtered` zurückgegeben.

16. Filterung einer WAV Datei in Micropython

Zur Demonstration von digitalen Filtern in Micropython wird eine WAV-Datei mittels eines ESP32 gefiltert. Die Audio-Datei wird über eine SD-Karte eingelesen, gefiltert und als gefilterte Variante abgespeichert.

Zu beachten ist, dass nur 16-Bit-WAV-Dateien unterstützt werden. Die Datei kann jedoch Mono oder Stereo mit und einer beliebigen Samplerate sein. Die WAV-Datei, die gefiltert werden soll, muss zu `input.wav` umbenannt werden bevor diese auf die SD-Karte übertragen wird.

16.1. Vorbereitung des Programms

```
import micropython, struct, os,
from machine import SDCard, freq
from biquad import BiquadFilterTDF2
```

Für die Filterung einer WAV-Datei werden die folgenden Module eingebunden: `micropython` ermöglicht die Optimierung durch native Deklaration, `struct` dient der binären Datenverarbeitung, `os` wird für Dateizugriffe genutzt und `SDCard` stellt die Schnittstelle zur SD-Karte bereit, während mit `freq` die CPU-Frequenz gesetzt werden kann. Aus der erstellten `biquad.py` wird die `BiquadFilterTDF2` importiert.

```
freq(240000000)
```

Die CPU-Frequenz wird auf 240 MHz gesetzt, um maximale Rechenleistung des ESP32 für die Filterverarbeitung zu gewährleisten.

```
os.mount(SDCard(slot=1, width=4), "/sd")
```

Die SD-Karte wird über das SPI-Interface mit 4-Bit Breite gemountet, um höhere Datenübertragungsraten zu ermöglichen.

16.2. Konfiguration der Audioverarbeitung

Audio-Parameter

```
INPUT = "/sd/input.wav"
OUTPUT = "/sd/output_filtered.wav"
HEADER_SIZE = 44
CHANNELS = 2
BITS = 16
BYTES_PER_SAMPLE = BITS // 8
BYTES_PER_FRAME = CHANNELS * BYTES_PER_SAMPLE
BLOCK_FRAMES = 13200  # Je nach RAM anpassen
BLOCK_SIZE = BLOCK_FRAMES * BYTES_PER_FRAME
```

Die Konstanten definieren die Audioparameter für Stereo-WAV-Dateien mit 16-Bit Auflösung. Die `BLOCK_FRAMES` beeinflussen den Speicherverbrauch vom RAM und die Verarbeitungseffizienz. Der `HEADS_SIZE` von 44 Bytes entspricht der Standardgröße eines WAV-Headers.

Filter-Instanziierung

```
b = [0.07033, -0.1380, 0.07033]
a = [1.00000, -0.1380, -0.8593]
gain = 1.0
filtL = BiquadFilterTDF2(b, a, gain)
filtR = BiquadFilterTDF2(b, a, gain)
```

Für die Filterung vom Stereo-Format werden zwei Filterinstanzen, filtL für den linken Kanal und filtR für den rechten Kanal, erstellt.

16.3. WAV-Header Erstellung

```
def write_header(f, frames):
    rate = 44100
    byte_rate = rate * CHANNELS * BYTES_PER_SAMPLE
    align = CHANNELS * BYTES_PER_SAMPLE
    size = frames * align
    f.write(struct.pack("<4sI4s4sIHHIIHH4sI",
        b'RIFF', 36 + size, b'WAVE',
        b'fmt ', 16, 1, CHANNELS, rate,
        byte_rate, align, BITS,
        b'data', size
    ))
```

Die `write_header()` Funktion generiert einen standardkonformen WAV-Header im Little-Endian Format. Der Header enthält alle notwendigen Informationen: Dateigröße, Audioformat(PCM), Kanalanzahl, Abtastrate und Bit-Tiefe. Das `struct.pack()` mit `"<4sI4s4sIHHIIHH4sI"` Format sorgt für die korrekte Byte-Reihenfolge auf verschiedenen Plattformen.

16.4. Blockverarbeitung

```
@micropython.native
def process_block_native(read_mv, write_mv, sample_count, filtL, filtR):
    for i in range(0, sample_count, 2):
        # Samples einlesen (Little Endian)
        sL = read_mv[i*2] | (read_mv[i*2 + 1] << 8)
        sR = read_mv[i*2 + 2] | (read_mv[i*2 + 3] << 8)
        if sL >= 32768:
            sL -= 65536
        if sR >= 32768:
            sR -= 65536
```

Zur Optimierung wird mit dem `@micropython.native` Decorator die Verarbeitungsschleife zu nativem Maschinencode kompiliert. Die Verwendung von `memoryview` Objekten vermeidet Speicherkopien und reduziert die Garbage Collection. Die Schleife iteriert in 2er-Schritten durch `sample_count`, die jeweils ein Stereo-Sample verarbeitet wird.

Sample-Konvertierung

Die 16-Bit-Samples werden aus dem Byte-Array im Little-Endian Format rekonstruiert. Die bitweise OR-Operation kombiniert die Low- und High-Bytes: das erste Byte wird als niederwertige Bits verwendet, das zweite Byte wird um 8 nach links verschoben für die höherwertigen Bits. Die anschließende Überprüfung konvertiert unsigned 16-Bit Werte in signed Werte durch die Subtraktion von 65536 für Werte über 32767.

16.5. Filterung und Clamping

```
    l = int(filtL.filter(sL))
    r = int(filtR.filter(sR))

    if l > 32767: l = 32767
    elif l < -32768: l = -32768
    if r > 32767: r = 32767
    elif r < -32768: r = -32768

    struct.pack_into("<hh", write_mv, i*2, l, r)
```

Bei der Filterung werden die Samples auf ganze Zahlen gerundet und auf den gültigen 16-Bit Bereich begrenzt. Das Clamping verhindert Überläufe und damit verbundene Verzerrung im Ausgangssignal. Die Begrenzung ist notwendig, da die Filteroperation Werte außerhalb des ursprünglichen Bereichs erzeugen kann. Abschließend werden die verarbeiteten Samples mit `struct.pack_into()` direkt in den Ausgabepuffer geschrieben. Das Format `"<hh"` definiert zwei signed 16-Bit Integers im Little-Endian Format.

16.6. Hauptverarbeitung

```
    read_buf = bytearray(BLOCK_SIZE)
    write_buf = bytearray(BLOCK_SIZE)
    read_mv = memoryview(read_buf)
    write_mv = memoryview(write_buf)
```

Separate Read- und Write-Puffer ermöglichen effiziente Ein- und Ausgabeoperationen ohne Interferenz. Die `memoryview` Objekte bieten direkten Zugriff auf die Pufferdaten ohne Kopieroperationen, wodurch die Performance verbessert wird.

```
with open(INPUT, "rb") as fin, open(OUTPUT, "wb") as fout:
    fin.seek(HEADER_SIZE)
    write_header(fout, 0)

    while True:
        read_bytes = fin.readinto(read_buf)
        if read_bytes == 0:
            break

        sample_count = read_bytes // 2  # 2 Bytes pro Sample (16 Bit)

        # Verarbeitung mit native-Code
        process_block_native(read_mv, write_mv, sample_count, filtL, filtR)

        fout.write(write_mv[:read_bytes])
        frames += read_bytes // BYTES_PER_FRAME
        last = progress(frames, total, last)
```

Über den `with` Context Manager wird die Eingangsdatei geöffnet. Zunächst wird der WAV-Header der Eingangsdatei mit `fin.seek(HEADER_SIZE)` übersprungen und ein temporärer Header mit 0 Frames in die Ausgabedatei geschrieben. Die Verarbeitungsschleife liest die Datenblöcke in den Eingangspuffer mit `readinto()`, was Speicherallokationen vermeidet. Bei `read_bytes = 0` wird das Dateiende erreicht. Der `sample_count` wird durch die Division der gelesenen Bytes durch 2 gerechnet, da jedes 16-Bit Sample 2 Bytes belegt. Nach der nativen Filterung wird der Ausgabepuffer in die Datei geschrieben, wobei `write_mv[:read_bytes]` sicherstellt, dass nur die gefilterten Bytes in die Ausgangsdatei geschrieben werden.

```
    # Header nachträglich korrigieren
    fout.seek(0)
    write_header(fout, frames)
```

Nach Abschluss der Verarbeitung wird der Header mit der korrekten Anzahl verarbeiteter Frames überschrieben. Dieser Schritt ist notwendig, da die finale Dateigröße erst nach vollständiger Verarbeitung bekannt ist. Mit `seek(0)` wird der Dateizeiger auf den Anfang der Header-Aktualisierung positioniert.

17. Echtzeit-Audiofilterung für Mikrofoneingang

Zur Demonstration der Echtzeitfilterung mit biquadratischen Filtern wird ein Mikrofonsignal digital gefiltert. Die Filterung wird mit einem ESP Lyrat 4.3 durchgeführt und mithilfe der audio-tools sowie den audio-board-driver Bibliotheken wird das Programm verfasst. Die `Filter.h` wurde für die Nutzung der transponierten Direktform 2 modifiziert, indem diese innerhalb des Bibliotheksverzeichnisses ausgetauscht wird.

17.1. Codeerklärung der Echtzeitfilterung

```
#include "AudioTools.h"
#include "AudioTools/AudioLibs/AudioBoardStream.h"
```

Für die Echtzeitfilterung werden die `AudioTools.h` und die `AudioBoardStream.h` eingebunden. Die `AudioBoardStream.h` ermöglicht einen I2S-Stream zwischen dem Audio-Codec des LyRaT mit dem ESP32 und die `AudioTools` ermöglichen die Verarbeitung von diesem Audiostream.

```
AudioInfo info(44100, 2, 16);
AudioBoardStream lyrat(LyratV43);
FilteredStream<int16_t, float> filtered(lyrat, info.channels);
StreamCopy copier(lyrat, filtered);
```

Die Audioparameter werden durch `AudioInfo` mit 44,1kHz, Stereo und 16-Bit definiert. Durch den Parameter `LyratV43` in `AudioBoardStream` wird die Boardinitialisierung vorbereitet. Der gefilterte Stream `filtered` nutzt den `lyrat` Stream als Eingang und wird in Stereo gefiltert. Der `StreamCopy copier` übernimmt die kontinuierliche Übertragung der gefilterten Audiodaten vom Eingangssignal zur Audioausgabe des LyRaT.

```
const float b[] = {1.0, 0.0, 0.0};
const float a[] = {1.0, 0.0, 0.0};
const float gain = 1.0;
```

Die Filterkoeffizienten werden durch drei Arrays definiert. `b[]` stellt die Zählerkoeffizienten des Biquads dar und `a[]` enthält die Nennerkoeffizienten. Der Verstärkungsfaktor wird durch `gain` festgelegt. Mit diesen Werten wird der Filter implementiert, welcher den I2S-Stream verarbeiten soll.

```
void setup(void) {
    Serial.begin(115200);
    filtered.setFilter(0, new BiQuadTDF2<float>(b, a, gain));
    filtered.setFilter(1, new BiQuadTDF2<float>(b, a, gain));
    auto config = lyrat.defaultConfig(RXTX_MODE);
    config.input_device = ADC_INPUT_LINE1;
    lyrat.begin(config);
}
```

Die Setup-Funktion beginnt mit der Initialisierung der seriellen Kommunikation mit 115200 Baud. Anschließend wird der Audio-Stream des LyRaT mit der Standard-TX-Konfiguration initialisiert. Für beide Audiokanäle werden separate Biquad-Filter in transponierter Direktform 2 erstellt, wobei der linke Kanal über Index 0 und der rechte Kanal über Index 1 angesprochen wird. Der Audio-Stream des Lyrat wird initialisiert, wobei die Eingabequelle auf `ADC_INPUT_LINE1` gesetzt wird, um die Onboard-Mikrofone als Audioquelle zu verwenden.

```
void loop() {
    copier.copy();
}
```

In der Loop-Funktion wird kontinuierlich `copier.copy()` aufgerufen, um die gefilterten Audiodaten vom Eingang zur Ausgabe zu übertragen und eine unterbrechungsfreie Echtzeit-Verarbeitung zu gewährleisten.

17.2. Filterkaskadierung

Die Filterkaskadierung erfolgt durch die serielle Anordnung mehrerer Filterstufen. Jede Stufe kann durch Biquad-Filter oder einzelne Sektionen einer SOS-Matrix realisiert werden.

```
filtered.setFilter(0, new FilterChain<float, 3>({
                      new BiQuadTDF2<float>(b, a, gain),
                      new BiQuadTDF2<float>(b, a, gain),
                      new BiQuadTDF2<float>(b, a, gain)
}));

filtered.setFilter(1, new FilterChain<float, 3>({
                      new BiQuadTDF2<float>(b, a, gain),
                      new BiQuadTDF2<float>(b, a, gain),
                      new BiQuadTDF2<float>(b, a, gain)
}));
```

Bei der Filterinstanziierung werden die Filterstufen mittels `FilterChain<float, 3>`definiert. Die einzelnen Biquad-Stufen können dabei entweder identische Koeffizienten verwenden oder unterschiedliche Koeffizientensätze erhalten, um Filter höherer Ordnung zu implementieren.

18. Praktische Anwendung der Bilinear-Transformation

Die analogen Biquad Filter für die Anwendung der Bilineartransformation werden aus dem Experiment 4 des Analog Systems Lab Kit PRO entnommen.

18.1. Übertragungsfunktionen der analogen Biquad Filter

Tiefpass: $H_{TP}(s) = \frac{V_3}{V_i} = \frac{+H_0}{1+\frac{s}{w_0 Q}+\frac{s^2}{w_0^2}} = \frac{+H_0 \cdot w_0^2}{w_0^2+\frac{s w_0}{Q}+s^2}$

Hochpass: $H_{HP}(s) = \frac{V_1}{V_i} = \frac{H_0 \cdot \frac{s^2}{w_0^2}}{1+\frac{s}{w_0 Q}+\frac{s^2}{w_0^2}} = \frac{+H_0 \cdot s^2}{w_0^2+\frac{s w_0}{Q}+s^2}$

Bandpass: $H_{BP}(s) = \frac{V_2}{V_i} = \frac{-H_0 \cdot \frac{s}{w_0}}{1+\frac{s}{w_0 Q}+\frac{s^2}{w_0^2}} = \frac{-H_0 \cdot s w_0}{w_0^2+\frac{s w_0}{Q}+s^2}$

Bandsperre: $H_{BS}(s) = \frac{V_4}{V_i} = \frac{(1+\frac{s^2}{w_0^2}) \cdot H_0}{1+\frac{s}{w_0 Q}+\frac{s^2}{w_0^2}} = \frac{(w_0^2+s^2) \cdot H_0}{w_0^2+\frac{s w_0}{Q}+s^2}$

18.2. Analoge Filterschaltung

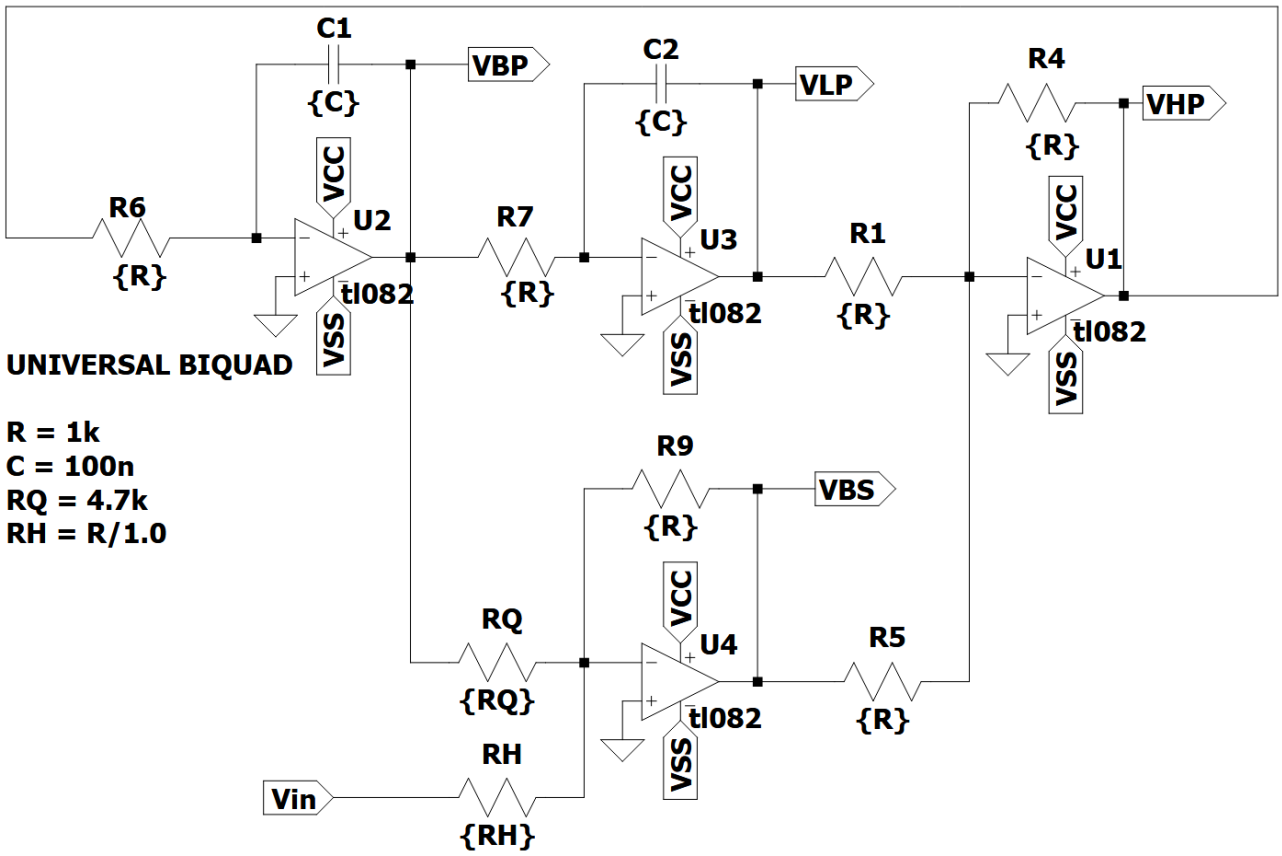


Abbildung 18.1.: Analoge Biquad Schaltung

Um diese analogen Filter Bilinear zu transformieren stehen MATLAB sowie Python zur Verfügung, um die Durchführung der Substitutionen durchzuführen.

18.3. Python

Python mit scipy.signal

```
numz, denz = bilinear(nums ,dens ,fs = fs)
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import freqs, bilinear, freqz
```

Für die Bilinear-Transformation wird die Scipy-Bibliothek mit dem Signal-Modul verwendet. Aus diesem Modul werden die Funktionen freqs, bilinear und freqz importiert. Für weitere Berechnungen wird Numpy als np importiert und für die Möglichkeit zum Plotten wird Matplotlib.pyplot als plt importiert.

```
R = 1000
C = 100e-9
w0 = 1 / (R * C)
Q = 4.7
fs = 44100
```

Aus dem Schaltbild der analogen Filterschaltung werden dessen Parameter entnommen. Die Kreisfrequenz w_0 berechnet sich durch $w_0 = \frac{1}{RC}$. Der Wert H_0 beschreibt den Gainfaktor der Filterschaltungen und wird auf 1 gesetzt, wodurch dieser nicht mehr im Code aufkommt. Die digitalen Filter sollen für Audioanwendungen verwendet werden, weshalb die Wahl von 44,1 kHz als Abtastfrequenz getroffen wurde. Durch die Wahl der Abtastefrequenz von 44,1 kHz wird kein Prewarping benötigt, da die Frequenzverzerrung vernachlässigbar ist.

```
# Tiefpass Zähler
TP_nums = [0, 0, w0**2]

# Hochpass Zähler
HP_nums = [1, 0, 0]

# Bandpass Zähler
BP_nums =  [0, -w0, 0]

# Bandstop Zähler
BS_nums = [1, 0, w0**2]

# Nenner
dens = [1, w0/Q, w0**2]
```

Die Zähler und Nenner der Übertragungsfunktionen der analogen Filter werden als Listen übernommen. Die Listenelemente entsprechen den Koeffizienten in absteigender Potenzordnung: $[s^2, s^1, s^0]$. Die Nenner der vier Filter sind identisch und müssen nur einmal übernommen werden.

```
# Bilineartransformation

# Tiefpass
TP_numz, TP_denz = bilinear(TP_nums, dens, fs = fs)

# Hochpass
HP_numz, HP_denz = bilinear(HP_nums, dens, fs = fs)

# Bandpass
```

```
BP_numz, BP_denz = bilinear(BP_nums, dens, fs = fs)

# Bandstop
BS_numz, BS_denz = bilinear(BS_nums, dens, fs = fs)
```

Die Bilineartransformation wird durch `bilinear` durchgeführt, indem dieser Funktion die Zähler- und Nennerlisten der jeweiligen Filter mit der Abtastfrequenzen f_s übergeben werden. Die Funktion liefert die Zähler- und Nennerkoeffizienten der digitalen Übertragungsfunktion.

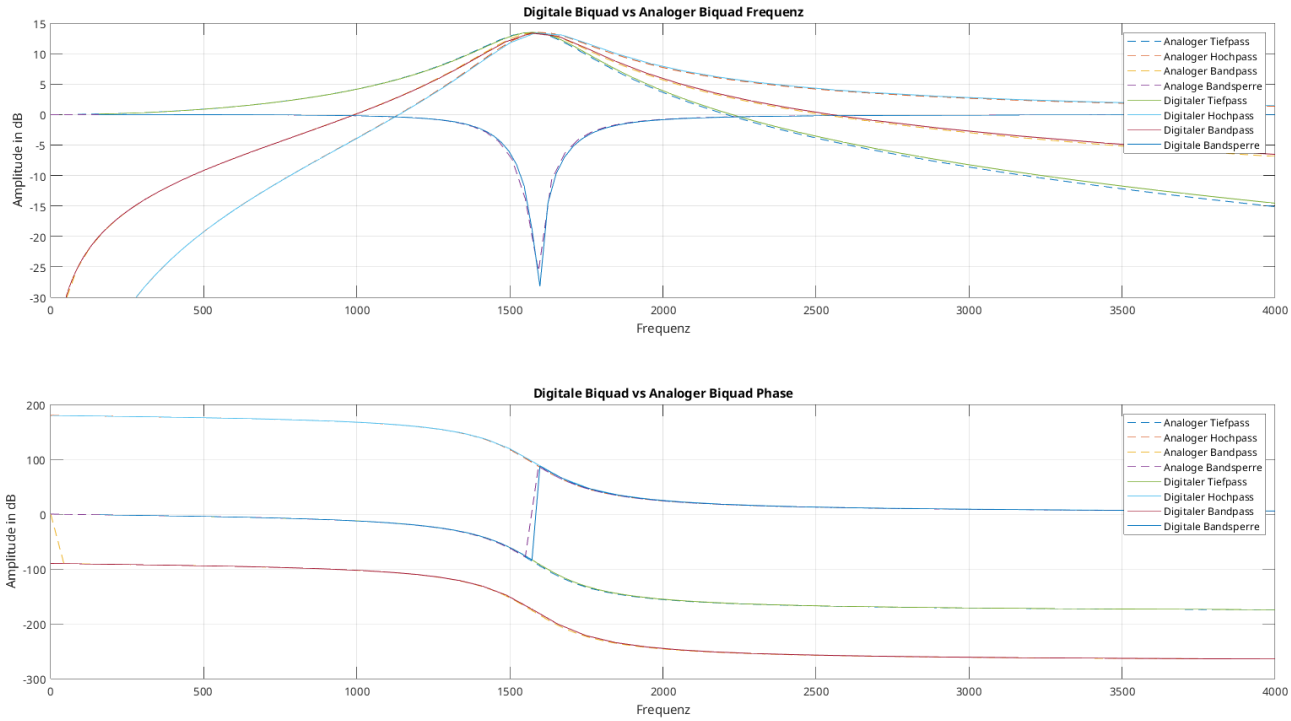


Abbildung 18.2.: Vergleich zwischen analogen und digitalen Frequenz- und Phasengang

19. Filterentwurf in Python

Für den Entwurf von digitalen IIR-Filtern wird in Python die Scipy-Bibliothek mit dem Signal-Modul verwendet. Diese Bibliothek bietet über eine umfassende Sammlung von Funktionen für den Entwurf und die Analyse von digitalen Filtern. Für den Entwurf von digitalen Biquad-Filtern wird das Entwurfsverfahren mit analogem Prototyping durchgeführt.

```
from scipy.signal import butter, cheby1, cheby2, ellip
```

Das Entwurfsverfahren basiert auf der Transformation analoger Filterprototypen, bei welchem ein analoger Filter mit den gewünschten Eigenschaften entworfen wird und anschließend durch die Bilineartransformation mit Prewarping in den digitalen Bereich überführt wird.

Zur beispielhaften Nutzung der Funktionen werden die vier Grundfiltertypen nach den folgenden Parametern entworfen:

```
# Ordnung für Tief- und Hochpass
order = 4
# Ordnung für Bandpass und Bandsperre
order_b = 2
```

Zur Demonstration werden Filter vierter Ordnung entworfen, um die Unterschiede zwischen der Filterprotypen zu verdeutlichen.

Für den Entwurf von biquadratischen Tief- und Hochpass-Filter wird die Ordnung auf den Wert `2` gesetzt. Bei Bandpass und Bandsperr-Filtern muss die Ordnung auf `1` gesetzt werden, um einen biquadratischer Filter zu erhalten.

Beim Entwurf von Bandpass- und Bandsperr-Filtern wird eine Frequenztransformation eines Tiefpass-Prototyps durchgeführt, bei welcher jede s-Variable quadriert wird, wodurch sich die Filterordnung automatisch verdoppelt.

```
# Abtastrate für Audioanwendungen in Hz - Standard für CD-Qualität
fs = 44100

# Grenzfrequenz für Tief- und Hochpass in Hz
fc = 1000
# Normalisierte Grenzfrequenz
wn = fc/(fs/2)
```

```
# Untere und obere Grenzen für Bandpass und Bandsperre in Hz
low = 500
high = 2000
# Normalisierte untere und obere Grenzen
wn_b = [low/(fs/2), high/(fs/2)]
```

```
# Welligkeit(Ripple) im Durchlassbereich in dB
rp = 0.5
# Dämpfung im Sperrbereich in dB
rs = 80
```

Über die Funtkionen `butter`, `cheby1`, `cheby2` und `ellip` werden die dementsprechenden Filter enworfen. Über den Parameter `btype` wird entschieden, welcher Filtertyp entworfen wird durchs Einsetzen von `low`, `high`, `bandpass`und `bandstop`.

Die Koeffizienten können als separate `b` und `a` Arrays oder als eine Second Order System (SOS)-Matrix durch `output = 'sos'` ausgegeben werden. Mit der SOS-Matrix bietet sich die Möglichkeit auch Filter höherer Ordnung, als kaskadierte Biquad-Sektionen zu realisieren. Für den Entwurf von digitalen Filtern wird beim Parameter `analog = False` gesetzt.

Mit dem Parameter `rp` lässt sich die Welligkeit (Ripple) im Durchlassbereich von Chebyshev1- und Elliptischen Filtern kontrollieren und mit dem Parameter `rs` die Dämpfung im Sperrbereich von Chebyshev2- und Elliptischen Filtern.

Butterworth-Filter

Der Butterwoth-Filter zeichnet sich aus mit seiner maximalen Flachheit im Durchlassbereich ohne Welligkeit aus. Als Kompromiss weist dieser einen relativ langsamen Übergang vom Durchlass- zum Sperrbereich auf. Er bietet ein ausgewogenes Verhältnis zwischen Phasenverhalten und Flankensteilheit.

```
b, a = butter(N = order, Wn = wn, btype = 'low', analog = False)
b, a = butter(N = order, Wn = wn, btype = 'high', analog = False)
sos = butter(N = order_b, Wn = wn_b, btype = 'bandpass',
             analog = False, output = 'sos')
sos = butter(N = order_b, Wn = wn_b, btype = 'bandstop',
             analog = False, output = 'sos')
```

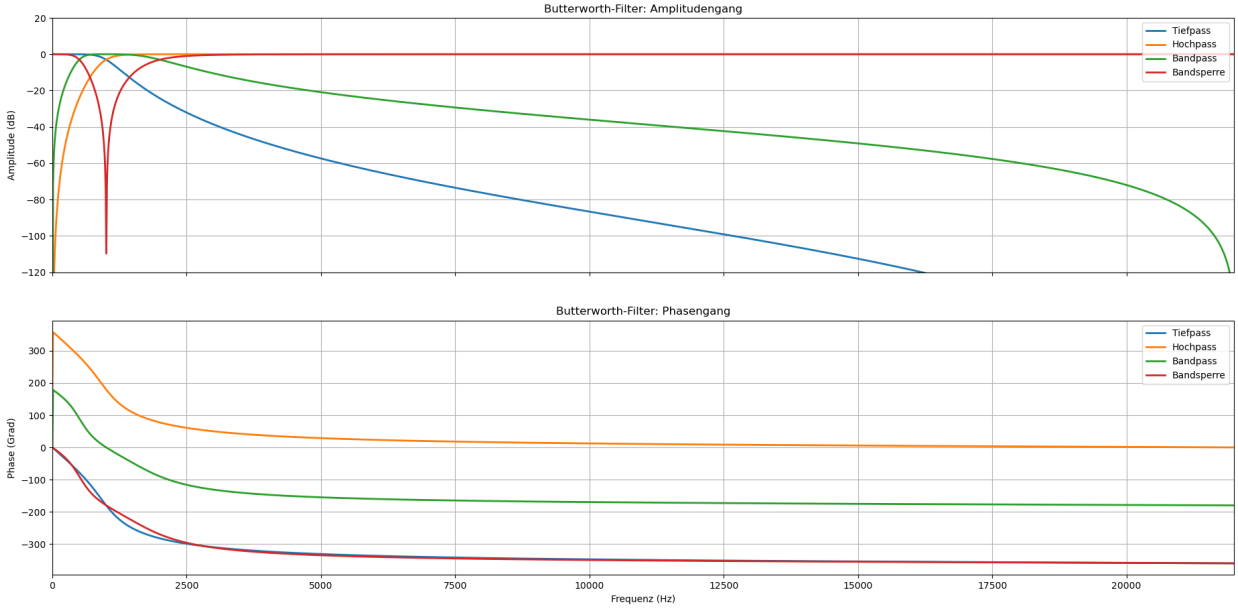


Abbildung 19.1.: Butterworth-Filter

Chebyshev Typ 1-Filter

Der Chebyshev Typ 1-Filter weist Welligkeit im Durchlassbereich auf, ermöglicht jedoch einen steileren Übergang im Vergleich zum Butterworth-Filter. Die Wellikgkeit ist gleichmäßig über den gesamten Durchlassbereich verteilt.

```
b, a = cheby1(N = order, rp = rp,Wn = wn, btype = 'low', analog = False)
b, a = cheby1(N = order, rp = rp, Wn = wn, btype = 'high', analog = False)
sos = cheby1(N = order_b, rp = rp, Wn = wn_b, btype = 'bandpass',
             analog = False, output = 'sos')
sos = cheby1(N = order_b, rp = rp, Wn = wn_b, btype = 'bandstop',
             analog = False, output = 'sos')
```

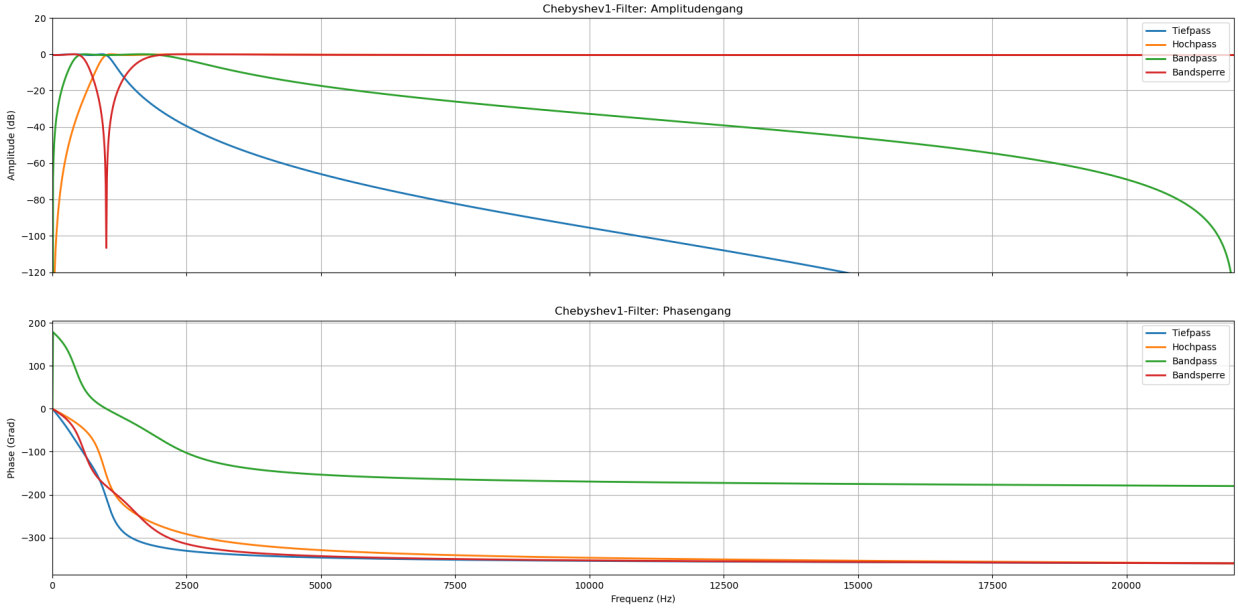


Abbildung 19.2.: Chebyshev1-Filter

Chebyshev Typ 2-Filter

Der Chebyshev Typ 2-Filter kombiniert einen flachen Durchlassbereich, vergleichbar mit dem Butterworth-Filter, mit Welligkeit im Sperrbereich. Bei gleichmäßiger Sperrbereichsdämpfung ermöglicht dieser steile Übergangsflanken.

```
b, a = cheby2(N = order, rs = rs,Wn = wn, btype = 'low', analog = False)
b, a = cheby2(N = order, rs = rs, Wn = wn, btype = 'high', analog = False)
sos = cheby2(N = order_b, rs = rs, Wn = wn_b, btype = 'bandpass',
             analog = False, output = 'sos')
sos = cheby2(N = order_b, rs = rs, Wn = wn_b, btype = 'bandstop',
             analog = False, output = 'sos')
```

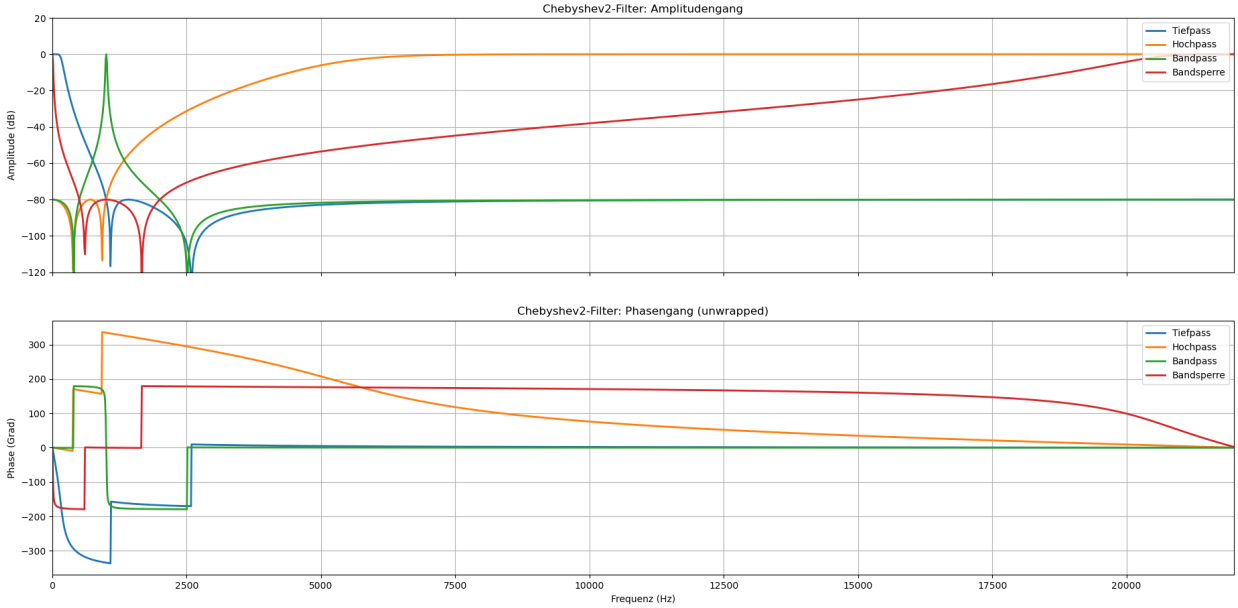


Abbildung 19.3.: Chebyshev2-Filter

Elliptischer Filter (Cauer)

Der elliptische Filter bietet die steilsten Übergangsflanken aller dargestellten Filtertypen, weist jedoch Welligkeiten sowohl im Durchlass- als auch im Sperrbereich auf. Dieser eignet sich Optimal für Anwendungen mit strengen Anforderungen an die Übergangsanforderungen.

```
b, a = ellip(N = order, rp = rp, rs = rs,Wn = wn, btype = 'low', analog = False)
b, a = ellip(N = order, rp = rp, rs = rs, Wn = wn, btype = 'high', analog = False)
sos = ellip(N = order_b, rp = rp,  rs = rs, Wn = wn_b, btype = 'bandpass',
            analog = False, output = 'sos')
sos = ellip(N = order_b, rp = rp,  rs = rs, Wn = wn_b, btype = 'bandstop',
            analog = False, output = 'sos')
```

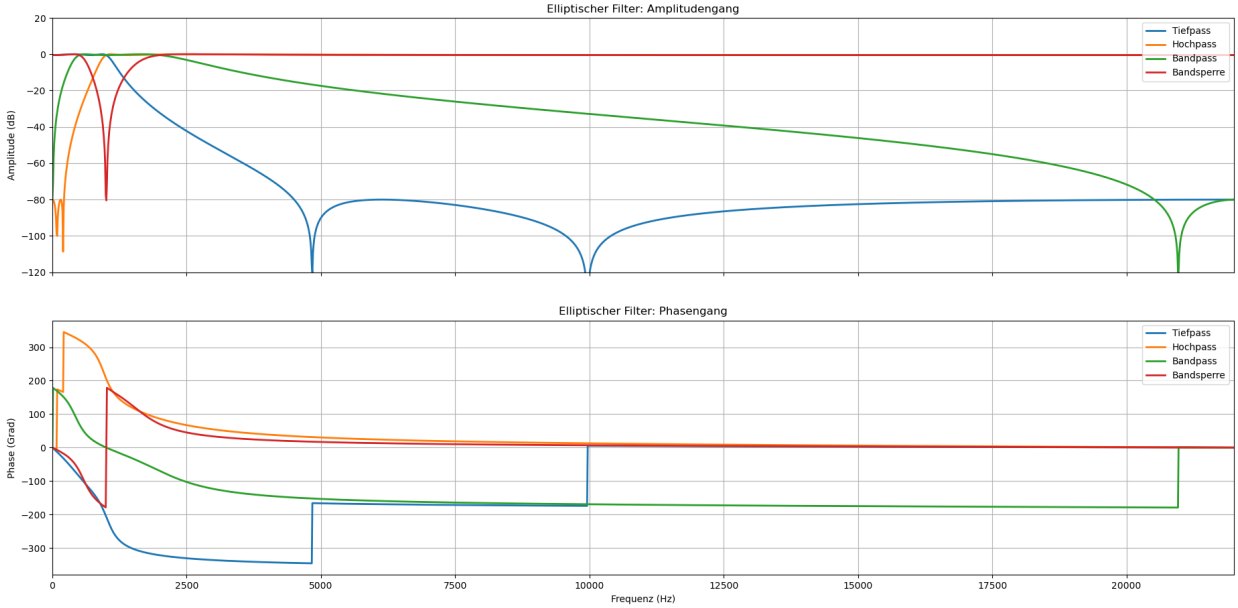


Abbildung 19.4.: Elliptischer Filter

Das Notebook `filter_desing.ipynb` ermöglicht eine Vergleichsanalyse der verschiedenen Entwurfsmethoden unter indentischen Parametern und bietet eine Basis für die Filterauswahl basierend auf den spezifischen Anwendungsanforderungen.

20. Pyfda - Pythonfilterdesigntool

Pyfda (Python Filter Design and Analysis) ist ein Open-Source-Tool, das als grafische Benutzeroberfläche für den Entwurf und die Analyse digitaler Filter entwickelt wurde. Die Anwendung basiert auf den Python-Bibliotheken PyQt5, NumPy, SciPy sowie Matplotlib und bietet Funktionalitäten, die dem Filter Design Tool in MATLAB ähneln.

Die Installation von Pyfda erfolgt über den Python Package Manager:

```
pip install pyfda
```

Das Tool wird anschließend mittels des Befehls `pyfdax` gestartet.

20.1. Demonstrativer Filterentwurf mit Frequenzgangbetrachung

Für die vorliegende Arbeit wurde ein Bandpassfilter 6. Ordnung mit einem Durchlassbereich von 300 Hz bis 3400 Hz entworfen. Diese Spezifikation entspricht dem Frequenzbereich der menschlichen Sprache und ist daher für Audioanwendungen von besonderer Relevanz.

Als Filterprototyp wurde ein elliptischer Filter (Cauer-Filter) gewählt, da dieser durch seine äquiripple-Charakteristik sowohl im Durchlass- als auch im Sperrbereich die steilsten Übergänge bei gegebener Filterordnung ermöglicht. Die Entwurfsparameter wurden wie folgt festgelegt:

- Welligkeit im Durchlassbereich: rp = 0.5 dB
- Dämpfung im Sperrbereich: rs = 90 dB
- Durchlassbereich: 300 Hz - 3400 Hz

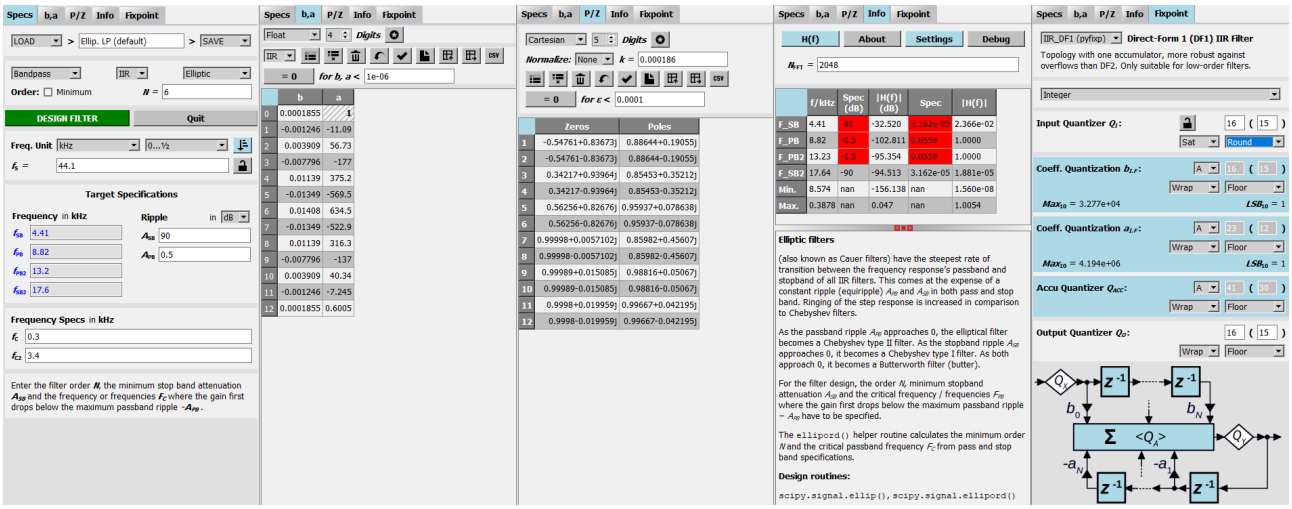


Abbildung 20.1.: Pyfda: Entwurfsparameter

Specs	**b, a**	**P/Z**	**Info**	**Fixpoint**
Spezifikation der Filterparamtern und des Entwurfsverfahrens.	Ausgabe der `b`- und `a`-Koeffizienten.	Ausgabe der Pole und Nullstellen.	Frequenzanalyse, Parameterauflistung sowie Beschreibung des Filtertyps.	Konvertierung zur Festkomma-Quantisierung für die Direktform 1.

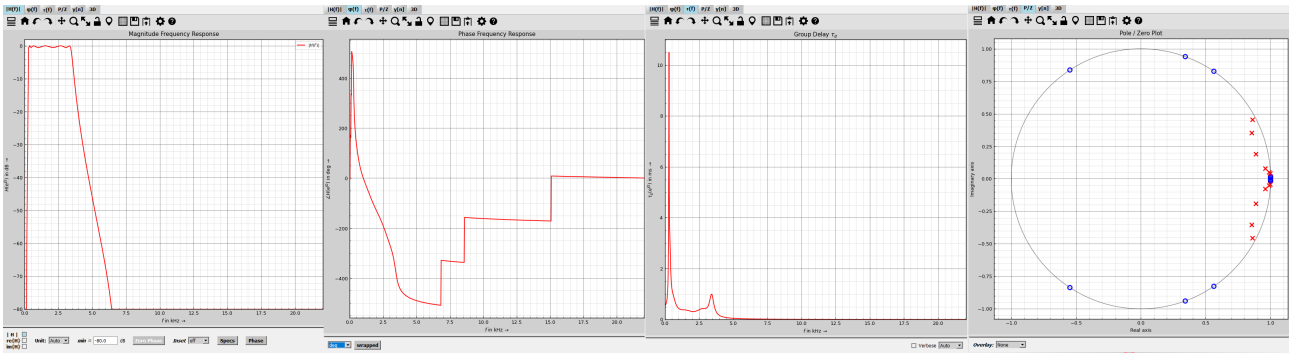


Abbildung 20.2.: Pyfda: Analyseplots

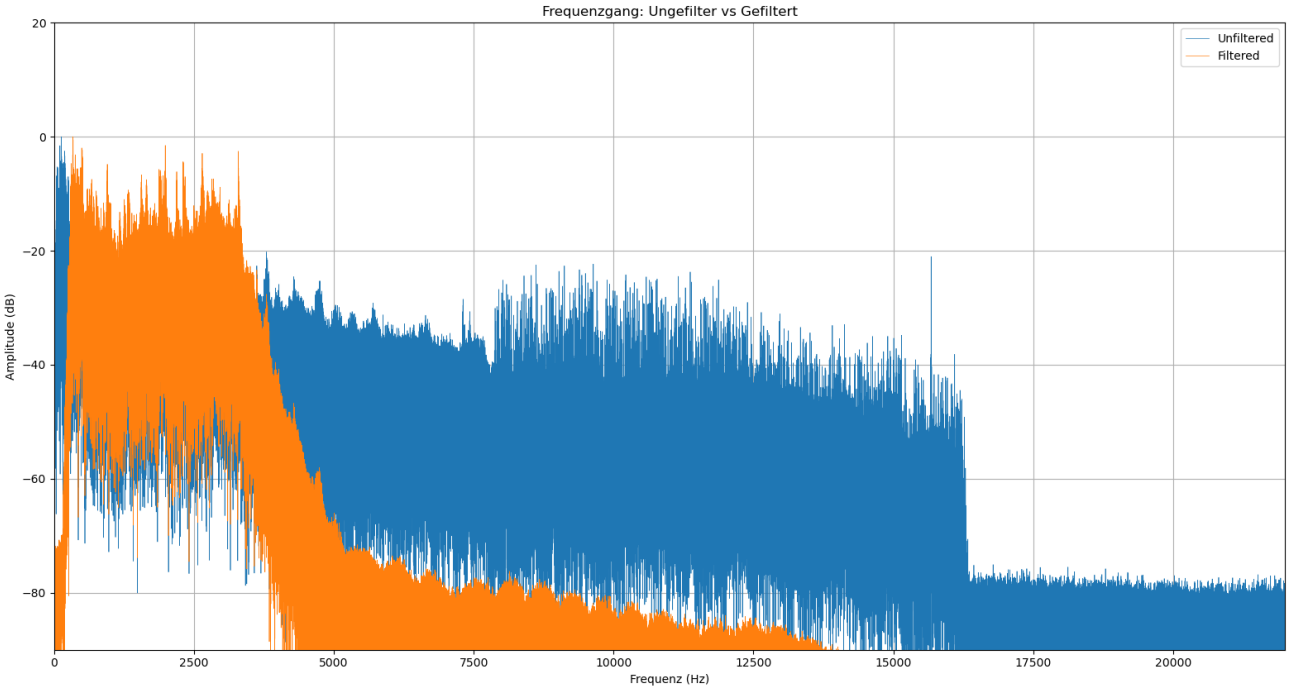
Magnitude Response	**Phase Response**	**Group Delay**	**Pole-Zero Plot**
Zeigt den Betrag der Übertragungsfunktion $\|H(f)\|$ in dB.	Zeigt die Phase von $H(f)$ über der Frequenz.	Zeigt die Gruppennachlaufzeit τ_g an.	Zeigt die Pole ($\times$) und Nullstellen ( ) in der z-Ebene.

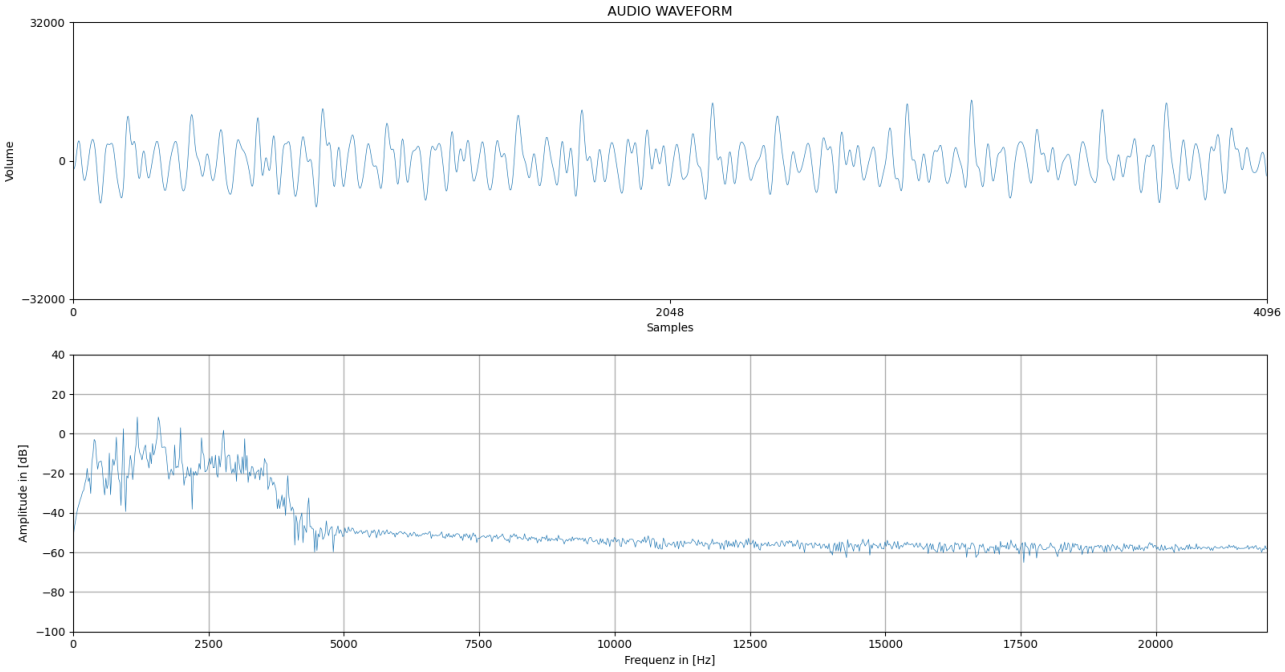
Obwohl Pyfda umfassende Entwurfs- und Analysemöglichkeiten bietet, beschränkt sich der Export auf die `b`- und ‘ae-Koeffizienten sowie die Pol- und Nullstellen. Für die Implementierung auf Mikrocontrollern wird jedoch eine (Second-Order-Sections) SOS-Matrix benötigt, die eine numerisch stabilere Darstellung des Filters ermöglicht.

```
sos = ellip(N = 6,
            rp = 0.5,
            rs = 90,
            Wn = [300/(44100/2], 3400/(44100/2)]
            btype = 'bandpass',
            analog = False,
            output = 'sos')
```

Diese SOS-Matrix kann anschließend in die entsprechenden Arduino-Implementierungen (`filter_wav_casceded.ino` oder `filter_microphone_cascaded.ino`) integriert werden.

Zur Veranschaulichung der Filterwirkung wurden Jupyter Notebooks entwickelt, die im Verzeichnis `filter_demonstration/visual/` verfügbar sind. Diese ermöglichen die Visualisierung von den Frequenzgängen sowohl der WAV-Dateien als auch der Echtzeitübertragung.




AUDIO WAVEFORM
32000
0
−32000
Volume
0
2048
4096
Samples
40
20
0
−20
−40
−60
−80
−100
Amplitude in [dB]
0
2500
5000
7500
10000
12500
15000
17500
20000
Frequenz in [Hz]

21. Evaluierung der Filterimplementierung auf Mikrocontrollern

21.1. Zielsetzung der Evaluierung

Ziel dieses Kapitels ist es, die praktische Umsetzung der digitalen Biquad-Filter hinsichtlich **Verarbeitungsdauer**, **Strukturauswahl**, **Stabilität** und **Echtzeitfähigkeit** systematisch zu bewerten. Dabei wird insbesondere der Vergleich zwischen MicroPython und Arduino untersucht. Darüber hinaus werden die verschiedene Filterstrukturen (Direktform 1, Direktform 2, transponierte Direktform 2) auf derselben Hardware verglichen. Abschließend wird die Verarbeitung eines IIR-Filters höherer Ordnung (6. Ordnung) implementiert und verarbeitet, um die Skalierbarkeit der Implementierung zu bewerten.

21.2. Testumgebung

Die Evaluierung wurde auf dem ESP32 basierten Lyrat V4.3 durchgeführt. Als Eingangssignal diente eine WAV-Datei mit folgenden Eigenschaften:

- **Dateiname:** `TF2_theme.wav`
- **Länge:** 1 Minute 12 Sekunden (72 Sekunden)
- **Abtastrate:** 44,1 kHz
- **Bittiefe:** 16 Bit
- **Kanäle:** Stereo
- **Gesamte Sample-Anzahl:** 3.215.360

Das Audiosignal wurde vollständig geladen und jeweils kanalweise (links/rechts) durch einen digitalen Filter verarbeitet. Die Messung der Laufzeit erfolgte mit `time.ticks_ms()` unter MicroPython. Für Vergleichbarkeit wurden je fünf Durchläufe pro Konfiguration durchgeführt.

21.3. Vergleich der TDF2: MicroPython vs. Arduino

Für den ersten Benchmark wurde eine Bandsperre zweiter Ordnung mit folgenden Designparametern implementiert:

- **Filterart:** IIR-Biquad (Bandsperre)

- **Fs:** 44,1 kHz
- **low cutoff:** 1250 Hz
- **high cutoff:** 1450 Hz
- **Koeffizienten:**
- `b = [0.07033, -0.1380, 0.07033]`
- `a = [1.00000, -0.1380, -0.8593]`

Dieser Filter wurde geziehlt grenzstabil Entworfen, um einen Vergleich zwischen der Stabilität der Strukturen zu Vergleichen.

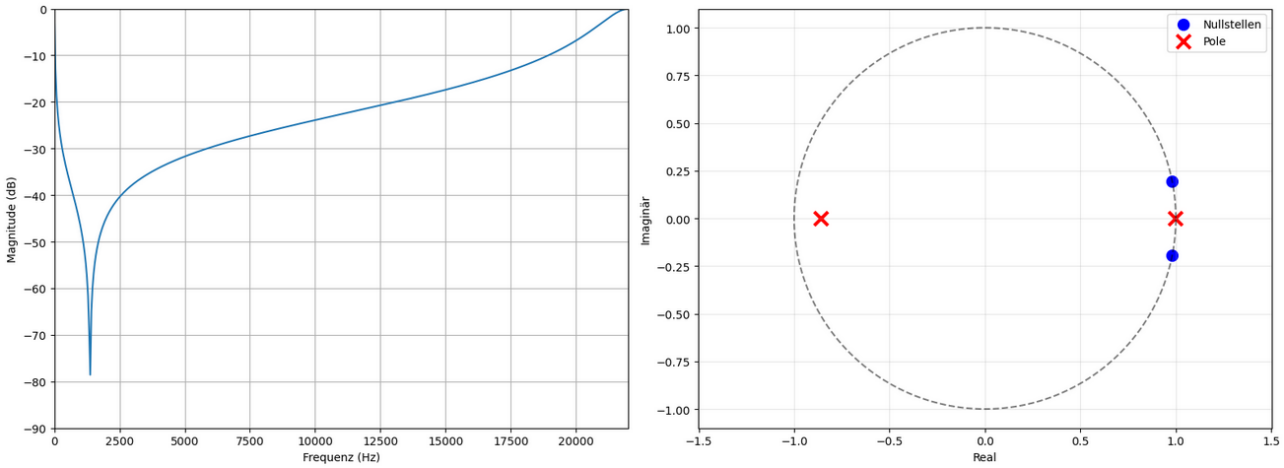


Abbildung 21.1.: Frequenzgang und Pol- und Nullstellen von der Bandsperre

21.3.1. Verarbeitungsdauer

Plattform	Struktur	Durchschnittliche Dauer	Spanne
MicroPython	TDF2	**549,27s**	548,66 – 550,69s
Arduino (C++)	TDF2	**16,60s**	16,48 – 16,97s

21.3.2. Interpretation

Die Ergebnisse zeigen eine massive Performance-Differenz zwischen den beiden Umgebungen: Die Arduino-Variante ist über 33-mal schneller als die MicroPython-Variante. Während die interpretierte Ausführung von MicroPython für einfache Skripte ausreichend ist, erweist sie sich bei Sample-genauer Signalverarbeitung als ungeeignet für Echtzeitanwendungen. Arduino bietet hingegen eine echtzeitnahe Ausführung mit direktem Hardwarezugriff und optimierter Kompilierung, wodurch die Umsetzung effizient und praxisnah gelingt.

21.4. Vergleich verschiedener Filterstrukturen auf Arduino

21.4.1. Übersicht der Filterstrukturen

- **Direktform 1 (DF1):** Klassische Struktur mit separater Verarbeitung von `x[n]` und `y[n]`.
- **Direktform 2 (DF2):** Speicheroptimierte Version mit gleichem Output, aber reduzierter Verzögerung.
- **Transponierte Direktform 2 (TDF2):** Numerisch stabilere Variante von DF2.

21.4.2. Verarbeitungszeitmessungen (Arduino)

Struktur	Verarbeitungszeiten (s)	Durchschnitt
TDF2	16,55 – 16,97	**16,60**
DF2	16,72 – 16,84	**16,78**
DF1	16,42 – 16,91	**16,53**

21.4.3. Funktionalität und Stabilität

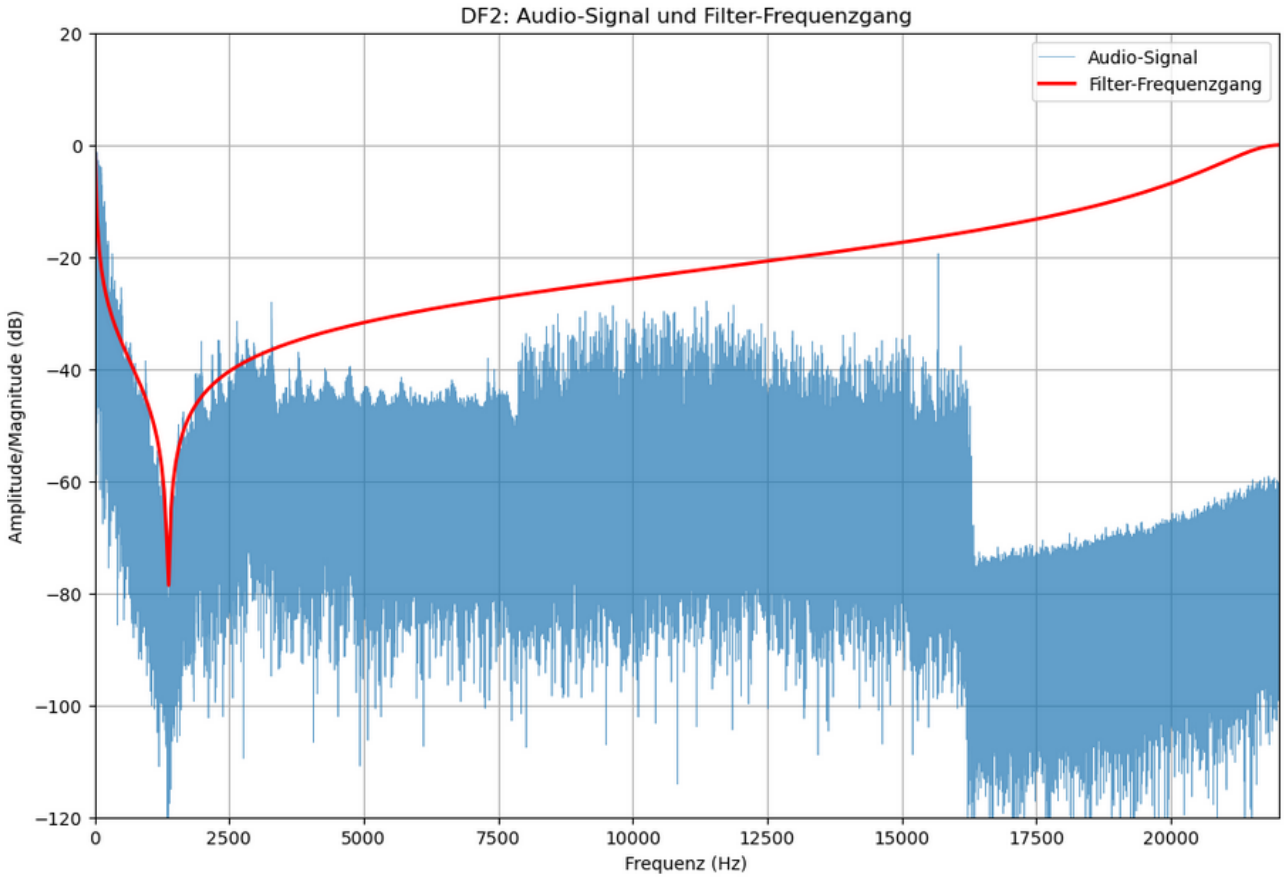
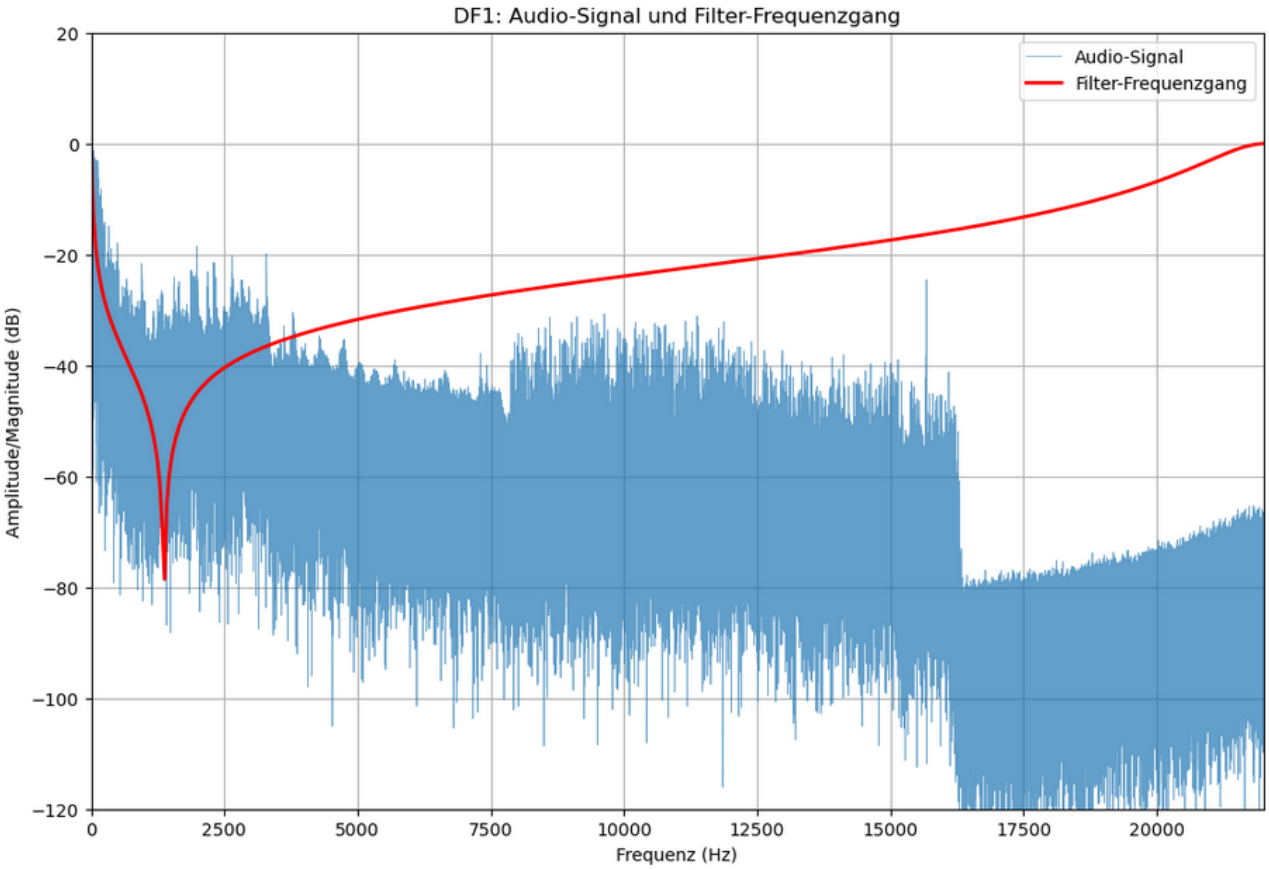


Abbildung 21.2.: DF2-Filterverlauf

Die Implementierungen von TDF2 und DF2 zeigten eine korrekte und deutlich wahrnehmbare Filterwirkung.



Die Implementierung mit DF1 führte hingegen nur zu einer minimalen Dämpfung, welche nicht dem Sperrbereich der Bandsperre entspricht. Eine Ursache könnte darin liegen, dass die Pole des Filters sehr nahe am Einheitskreis liegen. In Direktform 1 kann dies zu numerischer Instabilität und unzureichender Dämpfung führen, insbesondere bei Float-basierten Mikrocontrollerumgebungen mit begrenzter Genauigkeit.

21.4.4. Fazit Strukturen

Struktur	Durchschnitt (s)	Filterwirkung	Bewertung
TDF2	16,60	Eindeutig, stabil	Empfohlen für Echtzeitanwendung
DF2	16,78	Eindeutig, stabil	Ebenfalls gut geeignet
DF1	16,53	Nur leichte Dämpfung	Nicht geeignet bei kritischen Filtern

21.5. Erweiterung: Filterung mit sechster Ordnung (6 Biquad-Segmente in Kaskade)

21.5.1. Koeffizienten der Kaskade

Der entworfene Filter aus der PyFDA-Demonstration wurde für die Verarbeitszeitmessung zum Vergleich zwischen einfachen Biquads verwendet. Der Filter ist ein elliptischer Bandpass 6. Ordnung, wodurch die Filterung durch sechs Segmente ausgeführt wird.

```
Section 1: [b0, b1, b2, a0, a1, a2]
Section 2: [...]
...
Section 6: [...]
```

21.5.2. Ergebnisse (Arduino TDF2, 6 Biquad-Sektionen)

Lauf	Dauer (s)
1	23,50
2	23,47
3	23,45
4	23,48
5	23,43
Ø	**23,47**

21.5.3. Beobachtungen

- Im Vergleich zum einem einzelnen Biquad (16,60s) liegt die Verarbeitungszeitsteigerung bei ca. **41%**.
- Dies entspricht einer annähernd linearen Skalierung pro zusätzlicher Biquad-Stage.
- Auch bei höherer Ordnung bleibt der Filter **echtzeitfähig und speichereffizient**.

21.6. Zusammenfassung der Evaluierung

Aspekt	MicroPython	Arduino C++ (ESP32)
Filtergeschwindigkeit	Extrem langsam	Echtzeitnah (16s)
Strukturauswahl	Nur TDF2 praktikabel	TDF2 > DF2 > DF1
Stabilität (DF1)	n/a	Eingeschränkt bei bestimmten Koeffizienten
Erweiterbarkeit (Kaskade)	Ungeeignet	Problemlos bis 6. Ordnung

Aspekt	MicroPython	Arduino C++ (ESP32)
Entwicklungsaufwand	Gering	Mittel (C++)
Geeignet für Praxis?	Nein	Ja

Die durchgeführten Benchmarks bestätigen, dass die Implementierung digitaler Biquad-Filter auf dem ESP32 mit Arduino vollständig praxistauglich und echtzeitfähig ist. MicroPython hingegen ist für performante Audiobearbeitung ungeeignet, kann aber zum schnellen Prototyping sinnvoll sein. Die transponierte Direktform 2 stellt sich als beste Struktur durch ihre effizienz, stabilität und leichte skalierbarkeit heraus. Die erfolgreiche Anwendung eines Filters sechster Ordnung belegt zudem, dass auch komplexere Designs zuverlässig auf der Zielhardware ausführbar sind.

22. Gegenüberstellung von Mikrocontroller und FPGA Implementierung

Parallel zu dieser Bachelorarbeit wurde eine weitere Arbeit mit ähnlicher Fragestellung durchgeführt. Beide Arbeiten beschäftigen sich mit der praktischen Umsetzung digitaler IIR-Filter, unterscheiden sich jedoch grundlegend in der gewählten Implementierungsplattform. Während sich die eine Arbeit auf die softwarebasierte Realisierung mit FPGA konzentriert, steht in dieser Arbeit die softwarebasierte Realisierung mit Mikrocontrollern im Mittelpunkt.

Die Mikrocontroller-Implementierung erfolgt direkt in Arduino bzw. MicroPython. Die Biquad-Filter werden dabei auf Basis ihrer Differenzengleichungen manuell in Code umgesetzt. Verschiedene Strukturformen wie Direktform 1, Direktform 2 und deren transponierte Variante (TDF2) wurden explizit implementiert. Besonderheiten wie die manuelle Verwaltung interner Zustände oder Verzögerungsglieder sowie Optimierungen (z. B. durch `__slots__` in MicroPython oder native Kompilierung einzelner Methoden) tragen zur Effizienzsteigerung bei.

Während Arduino eine vergleichsweise leistungsstarke Umsetzung ermöglicht und eine Echtzeitverarbeitung über die I2S-Schnittstelle grundsätzlich mittles vorhandener Bibliotheken ermöglicht, ist MicroPython aufgrund seiner Interpreterstruktur deutlich langsamer. Zusätzlich verhinderten fehlende I2S-Treiber in MicroPython eine Audioverarbeitung in Echtzeit. Daher beschränkte sich die MicroPython-Implementierung auf die Filterung zuvor gespeicherter WAV-Dateien.

Im Gegensatz dazu basiert die FPGA-Implementierung auf einem modellbasierten Entwicklungsansatz mittels MATLAB/Simulink und dem HDL Coder. Die Filterstrukturen, insbesondere Direct Form II transponiert mit Pipelining, werden als Simulink-Modelle aufgebaut und anschließend in synthesefähigen VHDL-Code überführt. Die resultierenden IP-Cores werden über AXI4-Stream-Schnittstellen in ein Vivado-Projekt eingebunden. Die Verarbeitung erfolgt hardwareseitig, was eine hohe Parallelität und Echtzeitfähigkeit erlaubt. Die PYNQ-Plattform ermöglicht darüber hinaus eine komfortable Steuerung und Visualisierung über Jupyter Notebooks mit Python. Ein geplanter Echtzeitbetrieb über die I2S-Schnittstelle des integrierten Audio-Codecs ADAU1761 konnte jedoch aufgrund von Kompatibilitätsproblemen zwischen den automatisch generierten AXI-Strukturen und den bestehenden I2S-Komponenten nicht umgesetzt werden.

Insgesamt zeigen beide Arbeiten komplementäre Lösungswege auf. Die Mikrocontroller-Variante punktet durch ihre Einfachheit, Offenheit und geringe Einstiegshürden, während die FPGA-basierte Lösung eine leistungsstarke und skalierbare Architektur für komplexe Anwendungen bereitstellt. Beide Umsetzungen verdeutlichen unterschiedliche Strategien zur Realisierung digitaler Filter und bieten eine fundierte Grundlage für zukünftige Lehr-, Forschungs- oder Entwicklungsprojekte.

23. Reflexion

Während der Bearbeitung der Arbeit traten mehrere unerwartete Herausforderungen auf, die zu wertvollen Lernerfahrungen führten und die Bedeutung praxisnaher Umsetzung deutlich machten.

Ein wesentlicher Rückschlag betraf die geplante Echtzeitfilterung von Audiosignalen über Mikrofoneingänge mit MicroPython. Die Grundidee war es, eine einfache, interpretierbare Umgebung zur Signalverarbeitung bereitzustellen. In der praktischen Umsetzung stellte sich jedoch heraus, dass kein passender Treiber für den Audio-Codec (ES8388) in MicroPython verfügbar war. Aufgrund fehlender Vorkenntnisse in der Treiberentwicklung und begrenzter Zeit war es nicht möglich, den Codec in MicroPython in Betrieb zu nehmen. Somit konnte die geplante Mikrofonverarbeitung in dieser Umgebung nicht umgesetzt werden. Diese Einschränkung verdeutlicht, wie wichtig ein direkter Zugriff auf die Hardware ist und wo die Grenzen von stark vereinfachten Programmiersprachen wie MicroPython bei hardwarenahen Anwendungen liegen.

Ein weiterer Reflexionspunkt betrifft die unerwartet aufwendige WAV-Dateiverarbeitung. Es wurde ursprünglich davon ausgegangen, dass der WAV-Header der Eingabedatei einfach übernommen werden könne. In der Praxis stellte sich jedoch heraus, dass ein vollständig neuer Header generiert werden musste, da bestimmte Felder (wie Dateigröße, Datenlänge und Subchunks) dynamisch angepasst werden müssen. Dies erforderte eine tiefere Auseinandersetzung mit dem WAV-Dateiformat und führte zu einem deutlich längeren und komplexeren Codeabschnitt als geplant. Gleichzeitig bot dieser Teil der Arbeit jedoch auch die Möglichkeit, sich intensiv mit binärer Datenverarbeitung und Dateistrukturen auseinanderzusetzen.

24. Schlussbetrachtung

Im Rahmen dieser Bachelorarbeit wurde die vollständige Implementierung und Evaluierung digitaler Biquad-Filter auf einem ressourcenbeschränkten Embedded-System, dem Lyrat V4.3 erfolgreich durchgeführt. Ziel war es, grundlegende Kenntnisse der digitalen Signalverarbeitung, insbesondere der digitalen Filtertechnik, praxisnah anzuwenden und auf reale Audioverarbeitungssysteme zu übertragen.

Ausgehend von einer fundierten theoretischen Auseinandersetzung mit den Eigenschaften, Strukturen und Entwurfsverfahren digitaler Biquads wurden praxisrelevante Filter in Python entworfen und analysiert. Die Implementierung auf Mikrocontroller-Ebene erfolgte sowohl in der Arduino-Umgebung als auch in MicroPython, wobei verschiedene Filterstrukturen (Direktform 1, Direktform 2, transponierte Direktform 2) zum Einsatz kamen.

Die durchgeführte Evaluierung zeigt, dass die Arduino-Umgebung deutlich überlegen ist, was Verarbeitungszeit, Echtzeitfähigkeit und praktische Anwendbarkeit betrifft. Während MicroPython sich gut für das prototypische Testen eignet, ist sie für leistungsaufwendige Audiosignalverarbeitung in Echtzeit nicht geeignet. Die TDF2-Struktur erwies sich als die stabilste und effizienteste Form der Filterumsetzung auf Mikrocontrollern. Auch die erfolgreiche Realisierung eines Kaskadenfilters sechster Ordnung demonstrierte die Skalierbarkeit des Konzepts ohne signifikanten Verlust an Performance.

Mit dieser Arbeit wurde gezeigt, dass sich anspruchsvolle digitale Signalverarbeitung auch auf leistungsschwacher Embedded-Hardware effektiv realisieren lässt, vorausgesetzt, Strukturwahl, Implementierung und Filterdesign sind sorgfältig aufeinander abgestimmt. Durch den konsequenten Bezug zur Praxis, kombiniert mit methodischem Vorgehen und objektiver Bewertung, leistet diese Arbeit einen Beitrag zur anwendungsorientierten Vermittlung digitaler Filtertechnik im Bereich der Mikrocontroller und Open-Scource.

25. Ausblick

Die im Rahmen dieser Arbeit entwickelte Implementierung digitaler Biquad-Filter auf Embedded-System bildet eine solide Grundlage für weiterführende Arbeiten in der digitalen Signalverarbeitung. Im nächsten Schritt bietet sich insbesondere die Erweiterung auf FIR-Filter (Finite Impulse Response) an. Diese stellen eine numerisch stabile Alternative zu IIR-Filtern dar und eignen sich durch ihre lineare Phasenlage besonders für Anwendungen in der Audiotechnik.

Darüber hinaus ließe sich die theoretische Basis deutlich vertiefen, etwa durch Inhalte aus dem Masterstudium wie Multiraten-Systeme, Filterbänke, Spektralschätzverfahren oder adaptive Filteralgorithmen. Eine solche Erweiterung der Theorie würde nicht nur eine umfangreichere Analyse ermöglichen, sondern auch den Übergang zu komplexeren digitalen Signalprozessor-Systemen vorbereiten.

Ein langfristiger Entwicklungspfad liegt schließlich in der Integration analoger Filterstrukturen in integrierter Form, durch den Entwurf und die Simulation analoger Hochfrequenzfilter als ASICs oder auf FPGA-Basis mit analogen Schnittstellen. Die Verbindung der digitalen Entwurfsumgebung mit einer realen Hardwareimplementierung auf analoger Ebene würde die Brücke zwischen der rein digitalen Signalverarbeitung und der physikalischen Signalwelt schlagen und ein tiefes Verständnis für die Systemintegration im Bereich des Analog-Mixed-Signal-Designs fördern.

Part III.

FPGA

26. Biquads: Digitale Filter auf dem Pynq-Z2 als Lehrdemonstration

27. Abstract

Diese Arbeit befasst sich mit der Entwicklung, Simulation und FPGA-basierten Implementierung mehrerer digitaler IIR-Filter unter Verwendung der PYNQ-Plattform. Ziel ist die Realisierung von Filterarchitekturen, die sich für den Einsatz in einer interaktiven Lehrumgebung eignen. Im Mittelpunkt steht die Verbindung von MATLAB-gestütztem Filterentwurf, der Implementierung in VHDL sowie der Integration in eine Python-basierte Steuerungsumgebung über Jupyter Notebooks.

Die Implementierung der Filter erfolgt sowohl mit eigens erstellten VHDL-Komponenten über den HDL-Coder als auch mit vorkonfigurierten IP-Blöcken von Xilinx/AMD. Die Steuerung und Visualisierung werden mithilfe eines Jupyter-Notebooks umgesetzt, das auf dem ARM-Prozessor des PYNQ-Boards ausgeführt wird. Die funktionale Umsetzung der Filter konnte erfolgreich nachgewiesen werden. Im praktischen Betrieb traten jedoch Einschränkungen auf, welche die kontinuierliche Verarbeitung von Audiosignalen in Echtzeit verhinderten. Als alternative Lösung wurde das System erweitert, um zuvor aufgezeichnete Audiodaten zu verarbeiten. Zusätzlich wurde das Design so ergänzt, dass analoge Signale aufgenommen und ausgegeben werden können.

Besonderes Augenmerk gilt dem Einsatz von Biquad-Strukturen zur Realisierung der IIR-Filter. Die entwickelte Lösung wird abschließend mit einer alternativen Implementierungsvariante verglichen.

28. Motivation

Im Rahmen früherer Lehrveranstaltungen wurden bereits zahlreiche Erfahrungen mit analogen Filtern gesammelt. Auch die Grundlagen digitaler Filter wurden theoretisch behandelt und entsprechende Filterentwürfe in MATLAB simuliert. Eine tatsächliche Hardwareimplementierung digitaler Filter fand bisher jedoch nicht statt. Aus diesem Umstand stellt sich die Frage, wie digitale Filter in Hardware implementiert werden können.

Grundsätzlich stehen für die praktische Umsetzung digitaler Filter Plattformen wie Mikrocontroller und FPGAs (Field Programmable Gate Arrays) zur Verfügung. Im Rahmen dieser Arbeit soll die FPGA-basierte Umsetzung untersucht werden, da diese eine besonders leistungsfähige Alternative bietet. Konkret wird die Implementierung digitaler IIR-Filter auf dem FPGA-basierten PYNQ-Z2 Board untersucht. Diese Wahl ist bewusst getroffen worden, da Biquad-Filter bereits in analoger Form realisiert wurden und somit ein direkter Vergleich zwischen analoger und digitaler Implementierung möglich ist.

Zur Reduktion des Entwicklungsaufwands und zur Beschleunigung der Designphase wird der modellbasierte Ansatz mit MATLAB/Simulink und dem HDL Coder verwendet. Dieser Workflow ermöglicht eine automatische Übersetzung eines Simulink-Modells in synthesefähigen VHDL-Code. So kann die Methodik des Rapid Prototyping angewendet werden, um die entwickelten Filtermodule zügig zu testen und weiterzuentwickeln.

Die Implementierung über ein FPGA ist dabei stark an industrielle Vorgehensweisen angelehnt. Dies zeigt sich unter anderem in der Verwendung von kostenintensiven Lizenzprogrammen wie MATLAB sowie professionellen Entwicklungsumgebungen wie Vivado. Auch die benötigte Hardware, etwa FPGA-Entwicklungsboards, ist im Vergleich zu Mikrocontroller-Lösungen häufig deutlich kostenintensiver. Eine alternative Herangehensweise bietet die Implementierung über Mikrocontroller. Hier besteht die Möglichkeit, den gesamten Entwicklungsprozess mit kostenfreien Open-Source-Tools umzusetzen. Mikrocontroller selbst sind in der Regel ebenfalls deutlich günstiger als FPGA-basierte Systeme, was sie insbesondere für kostensensitive Anwendungen attraktiv macht.

Ziel der Arbeit ist es, die Implementierungsmethoden digitaler IIR-Filter auf einem FPGA exemplarisch zu demonstrieren und für den Lehreinsatz aufzubereiten. Die entwickelten Filter sollen auf dem PYNQ-Z2 einsatzbereit sein und als Grundlage für zukünftige Lern- und Demonstrationszwecke dienen.

29. Einleitung

Die kontinuierliche Weiterentwicklung der digitalen Signalverarbeitung sowie die steigende Leistungsfähigkeit rekonfigurierbarer Hardwarekomponenten ermöglichen zunehmend komplexere Implementierungen im Bereich digitaler Filterarchitekturen. Insbesondere in Anwendungen der Audiotechnik spielt die effiziente Verarbeitung von Signalen eine zentrale Rolle. Zur Erfüllung der wachsenden Anforderungen an Flexibilität, Rechenkapazität und Energieeffizienz geraten Field Programmable Gate Arrays (FPGAs) verstärkt in den Fokus aktueller Forschung sowie industrieller Applikationen.

Im Rahmen dieser Arbeit werden digitale IIR-Filter auf Basis eines Biquad-Ansatzes entworfen, simuliert und hardwareseitig auf einem FPGA innerhalb der PYNQ-Plattform implementiert. Ziel ist die Entwicklung einer Lehrdemonstration, die sowohl die technische Umsetzung als auch die Aufbereitung berücksichtigt. Der Fokus liegt auf dem Zusammenspiel aus MATLAB-basiertem Filterentwurf, High-Level-Synthese mittels VHDL und der Integration in eine Python-gesteuerte Umgebung. Eine zentrale Herausforderung stellt dabei die Entwicklung einer Architektur dar, die eine flexible Ansteuerung sowie eine anschauliche Visualisierung innerhalb der Jupyter-Notebook-Oberfläche erlaubt.

Zu Beginn werden die eingesetzte Hardwareplattform sowie die relevanten Grundlagen der digitalen Filterung und der verwendeten Entwicklungswerkzeuge vorgestellt. Darauf aufbauend erfolgt die Konzeption und Implementierung der digitalen Biquad-Filterstruktur mit anschließender Integration in die PYNQ-Umgebung. Abschließend wird die entwickelte Lösung hinsichtlich ihrer Eigenschaften und praktischen Anwendbarkeit evaluiert und mit einer alternativen Implementierungsvariante verglichen.

Diese Bachelorarbeit dokumentiert den Entwicklungs- und Implementierungsprozess, der im zugehörigen GitHub-Repository festgehalten wurde. Sowohl das Repository als auch Teile dieser Arbeit sind im Rahmen einer Zusammenarbeit mit einer parallel durchgeführten Bachelorarbeit entstanden, die eine vergleichbare Fragestellung behandelt. Da beide Arbeiten auf denselben theoretischen Grundlagen basieren, wurden diese gemeinsam erarbeitet. Auch der Vergleich der unterschiedlichen Implementierungsmethoden erfolgte in enger Abstimmung und wurde kollaborativ ausgearbeitet.

30. Das PYNQ-Z2 und die PYNQ-Plattform

Die Implementierung des Filters soll auf dem PYNQ-Z2 erfolgen. Das PYNQ-Z2 ist ein FPGA-Entwicklungsboard, das speziell für die Verwendung mit der PYNQ-Plattform konzipiert wurde. Die Plattform wurde von Xilinx (heute AMD) entwickelt und verfolgt das Ziel, die Entwicklung von FPGA-Anwendungen mithilfe von Python zu vereinfachen. Anstelle über herkömmlicher Hardwarebeschreibungssprachen ermöglicht PYNQ die Nutzung von Python-Code zur Steuerung und Nutzung programmierbarer Logik auf Zynq-SoCs, direkt über Jupyter Notebooks im Webbrowser. PYNQ steht für *Python Productivity* for Zynq und ist ein Open-Source-Projekt von AMD, das die Entwicklung von Anwendungen für Zynq-SoCs erheblich vereinfacht. Ziel der Plattform ist es, die komplexe Hardwareentwicklung mit FPGAs zugänglicher zu machen. Das Board wird typischerweise mit einer speziell angepassten Linux-Distribution für PYNQ von einer microSD-Karte gestartet. Über das Netzwerk lässt sich die integrierte Jupyter-Notebook-Oberfläche aufrufen, um direkt mit Python zu arbeiten und Hardwarefunktionen anzusteuern. Die Besonderheit von PYNQ liegt darin, dass komplexe Hardwaredesigns in der programmierbaren Logik per Overlay integriert und anschließend per Python angesteuert werden können. Dies ermöglicht beispielsweise Anwendungen in der Signalverarbeitung, beim maschinellen Lernen, in der Bildverarbeitung oder Robotik, ohne tiefgehende FPGA-Kenntnisse.

30.1. Hardware-Spezifikationen des PYNQ-Z2

Die vollständigen Hardware-Spezifikationen des Boards sind auf der offiziellen Produktseite bei AMD einsehbar.

Für die hier vorgestellte Implementierung sind insbesondere folgende Spezifikationen relevant:

PYNQ-Z2	Spezifikationen:
FPGA / SoC:	Xilinx Zynq-7000 SoC (XC7Z020-1CLG400C), 256 KB On-Chip-Memory, 630 KB Block RAM, 220 DSP-Slices
Audio-I/O:	I²S-Interface mit 24-Bit-DAC (3,5 mm TRRS-Buchse), Line-In (3,5 mm Klinke)
Netzwerk:	10/100/1000 Mbit/s Ethernet
Speicher:	512 MB DDR3-RAM (16-Bit-Bus, 1050 Mbps), 128 Mbit Quad-SPI-Flash, microSD-Karten-Slot
Taktquellen:	125 MHz für die programmierbare Logik (PL), 50 MHz für den Prozessor (PS)

30.2. ADAU1761 Audio-Codec

Auf dem PYNQ-Z2 ist ein ADAU1761 Audio-Codec von Analog Devices verbaut. In Kombination mit dem entsprechenden Audio-Modul aus der PYNQ-Bibliothek ermöglicht dieser Codec die direkte Aufnahme und Wiedergabe von Audiosignalen über die analogen Ein- und Ausgänge des Boards. Obwohl der ADAU1761 über integrierte DSP-Funktionen verfügt, werden diese nicht genutzt, da die Filterung vollständig in der programmierbaren Logik (PL) des FPGAs erfolgt. Ein besonderes Merkmal des ADAU1761 ist sein integrierter fraktionaler Phasenregelkreis (PLL). Dieser erlaubt es, aus einer Vielzahl externer Takteingänge (8 MHz bis 27 MHz) intern stabile Systemtakte zu erzeugen. Die vollständigen Spezifikationen sind im Datenblatt zu dem ADAU1761 Audio-Codec einsehbar.

ADAU1761	Spezifikationen:
Audioauflösung:	24 Bit (Sigma-Delta ADC/DAC)
Abtastraten:	8 kHz bis 96 kHz (standardmäßig 48 kHz)
Schnittstelle:	I²S (Inter-IC Sound) – digitale Audioverbindung in programmierbarer Logik
Steuerung:	Über I²C-Bus (in PYNQ über *pynq.lib.audio.AudioADAU1761*)
PLL-Taktgenerator:	Intern aus 8 MHz–27 MHz Eingangstakt generierbar

30.3. Verwendete Entwicklungswerkzeuge

Für die Entwicklung und Implementierung der digitalen Filter werden MATLAB mit Simulink sowie die Vivado Design Suite verwendet. Die Modellierung und Simulation des Filters erfolgt zunächst in Simulink, wobei der *HDL Coder* zur automatisierten Erzeugung des VHDL-Codes eingesetzt wird. Anschließend wird der generierte Filter in Vivado in das Gesamtsystem integriert, mit der AXI-Peripherie verbunden und der finale Bitstream für die PYNQ-Z2-Plattform erzeugt.

31. Filterentwurf in Matlab

In MATLAB wird der Entwurf digitaler IIR-Filter mit einer Vielzahl von Werkzeugen unterstützt, die sowohl den Entwurf als auch die Analyse und Implementierung erleichtern. Grundlage ist dabei das Prinzip, einen analogen Prototypfilter mittels Bilineartransformation in den digitalen Bereich zu überführen. Dieses Verfahren wird für gängige IIR Filtertypen wie Butterworth-, Chebyshev- und elliptische Filter eingesetzt. Die Funktionen *butter*, *cheby1*, *cheby2* und *ellip* wenden genau dieses Prinzip an. Die Vorgehensweise entspricht dabei der Filterentwicklung in Python mit der Scipy-Bibliothek.

Die direkte Umsetzung von IIR-Filtern mit Polynomkoeffizienten (b, a) kann bereits bei vergleichsweise niedrigen Ordnungen zu numerischen Instabilitäten führen. Ursache dafür sind Rundungsfehler, die durch die direkte Multiplikation aller Pole und Nullstellen im Übertragungsfunktionspolynom entstehen und dazu führen können, dass Pole außerhalb des Einheitskreises liegen. Dies beeinträchtigt die Stabilität und kann die Filterfunktion erheblich verfälschen. Um solche Instabilitäten zu vermeiden, werden IIR-Filter in der Praxis nicht als ein einziges großes Polynom, sondern als Produkt mehrerer einfacher biquadratischer Übertragungsfunktionen (Biquads) umgesetzt. Biquads sind IIR-Filterblöcke zweiter Ordnung, mit denen sich Filter höherer Ordnung durch Zerlegung und Kaskadierung in mehreren Sektionen stabil realisieren lassen. Diese Aufteilung begrenzt Rundungsfehler und verbessert die Gesamtstabilität des Filters. Dies ist insbesondere bei der Umsetzung auf Hardwareplattformen wie FPGAs oder Mikrocontrollern wichtig. MATLAB unterstützt dieses Vorgehen durch die Arbeit mit *Second-Order Sections (SOS)*. Die SOS-Darstellung wird als Matrix gespeichert, bei der jede Zeile die Zähler- und Nennerkoeffizienten einer Biquad-Sektion enthält. Zusätzlich wird ein separater Verstärkungsfaktor (*Gain*) ausgegeben, um den Frequenzgang korrekt zu skalieren. Die Entwurfsfunktionen können die Filterkoeffizienten direkt in SOS-Form bereitstellen. Alternativ lassen sich vorhandene Übertragungsfunktionen mit *tf2sos* in eine Biquad-Kaskade umwandeln.

Der Entwurfsprozess in MATLAB folgt dabei einem klaren Ablauf. Zunächst werden die Anforderungen festgelegt, wie Filtertyp, Filterordnung, Grenzfrequenzen und gegebenenfalls erlaubten Ripple im Durchlass- oder Sperrbereich. Die Grenzfrequenzen werden normiert auf die halbe Abtastrate (Nyquist-Frequenz) angegeben und müssen im Bereich von 0 bis 1 liegen. Bei Bandpass- und Bandsperrfiltern wird eine Frequenztransformation des Tiefpass-Prototyps durchgeführt, wodurch sich die Filterordnung mathematisch verdoppelt. Diese Verdoppelung muss bei der Wahl der Prototyp-Ordnung in Skripten berücksichtigt werden, da die Entwurfsfunktionen dies nicht automatisch anpassen.

Neben der rein skriptbasierten Arbeitsweise können Filter alternativ auch mit dem Filter Designer erstellt werden, der eine grafische Oberfläche für die Spezifikation der Filterparameter sowie die Visualisierung von Frequenzgängen und Pol-Nullstellen-Diagrammen bietet. Nach dem Entwurf lassen

sich die berechneten Filterkoeffizienten sowohl als Polynomkoeffizienten (b, a) als auch direkt in SOS-Form ausgeben. Die Kaskadierung in Biquads bezieht sich dabei stets auf die tatsächlich resultierende Gesamtordnung, einschließlich der Verdoppelung bei Bandpass- oder Bandsperrfiltern.

Für die Analyse stehen Funktionen wie *freqz* zur Darstellung von Amplituden- und Phasengang oder das umfassende Filter Visualization Tool (*fvtool*) zur Verfügung.

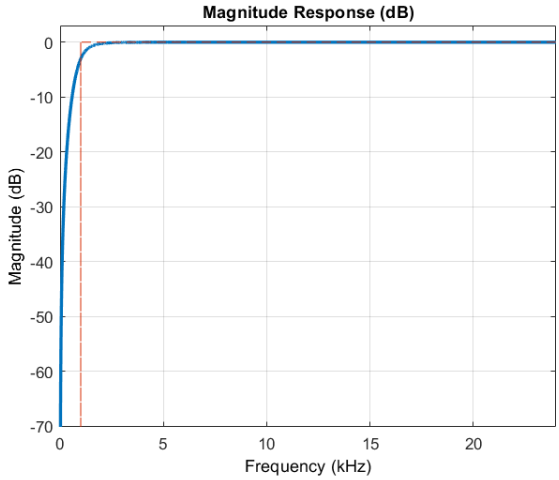
31.1. Spezifikation der Filter

Für die Demonstration wurde entschieden, alle vier grundlegenden Filtertypen als Butterworth-Filter zu realisieren. Diese besitzen den Vorteil kein Ripple im Durchlass- und Sperrbereich aufzuweisen. Gerade bei der Verarbeitung von Audiosignalen können Ripple im Frequenzgang zu unerwünschten Artefakten führen. Ein möglichst glatter Amplitudenverlauf ist daher insbesondere in der Audioanwendung vom Vorteil. Im direkten Vergleich zu elliptischen oder Chebyshev-Filtern weisen Butterworth-Filter bei gleicher Flankensteilheit eine höhere erforderliche Filterordnung auf. Diese höhere Ordnung würde in einer Hardwareumsetzung prinzipiell mehr Biquad-Stufen benötigen und damit mehr Ressourcen. Bei der Implementierung wurden die Filter auf 2. Ordnung beschränkt. Da bei einer so niedrigen Filterordnung alle Filter, unabhängig davon, ob sie als Butterworth- oder elliptischer Filter realisiert werden, ohnehin nur aus einer Biquad-Stufe bestehen, hätte die Wahl des Filtertyps keinen nennenswerten Unterschied bei den Hardwareanforderungen gemacht. Der Hauptgrund für die Entscheidung zugunsten der Butterworth-Filter liegt somit vor allem im glatten Frequenzverlauf.

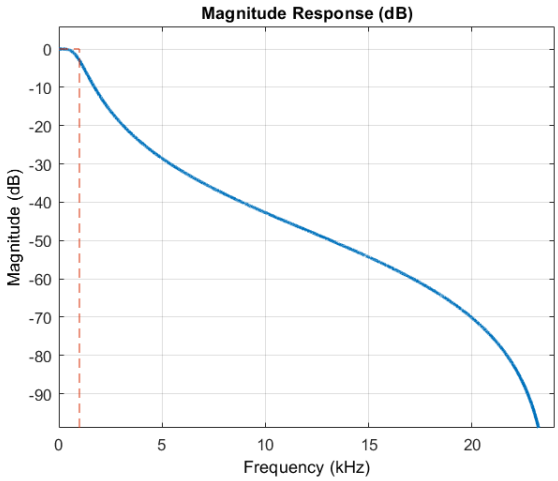
Bei der genauen Spezifikation der Filter wurde sich an dem Beispiel von Design of Digital Audio IIR Filters von Frank Schultz gehalten. Dabei ergaben sich Folgende Spezifikationen:

Filter:	Typ:	Fc:	Ordnung:
Hochpass	Butterworth	1kHz	2
Tiefpass	Butterworth	1kHz	2
Bandpass	Butterworth	500Hz - 2kHz	2
Bansstop	Butterworth	500Hz - 2kHz	2

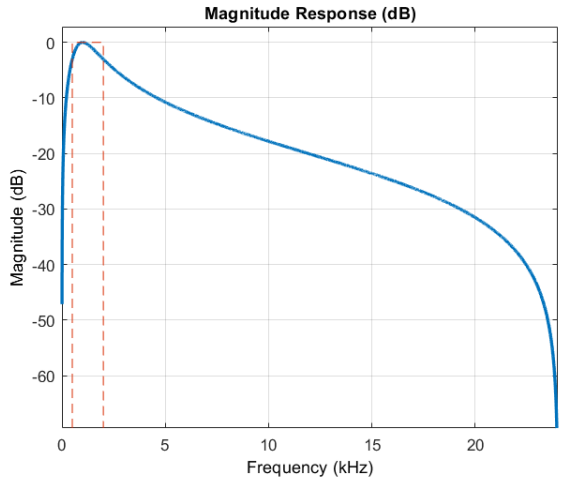
Alle Filter wurden mit dem Filter Designer auf einer Samplefrequenz von 48kHz entworfen und die Koeffizienten als SOS-Matrix inklusive Gain in das Workspace exportiert.



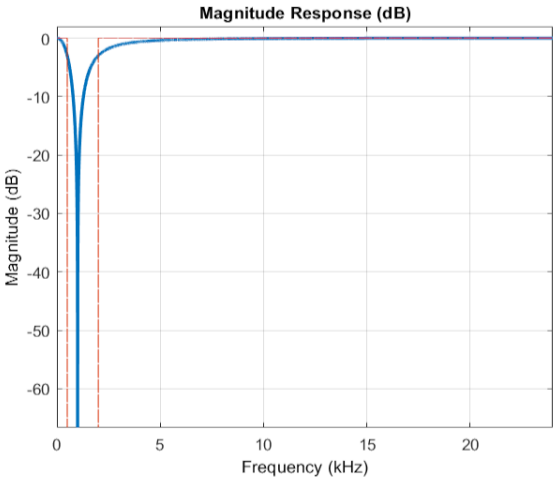
(a) Digitaler Butterworth-Hochpassfilter 2. Ordnung



(b) Digitaler Butterworth-Tiefpassfilter 2. Ordnung



(c) Digitaler Butterworth-Bandpassfilter 2. Ordnung



(d) Digitaler Butterworth-Bandstopfilter 2. Ordnung

Abbildung 31.1.: Digitaler Butterworth Filter: Amplitudengänge

32. Implementierung

Für die Implementierung von IIR-Filtern mittels VHDL gibt es mehrere Ansätze. Eine direkte Umsetzung der Differenzengleichung eines IIR-Filters in HDL-Code ist grundsätzlich möglich. Der Nachteil dieses Verfahrens liegt jedoch im Aufwand für Test, Validierung und Debugging. Gerade im Rahmen einer Abschlussarbeit mit begrenztem Zeitrahmen ist eine manuelle Umsetzung in dieser Tiefe schwer umsetzbar, zumindest nicht in einem vergleichbaren Umfang wie bei der Verwendung des HDL Coders von MATLAB/Simulink. Der modellbasierte Ansatz über MATLAB ermöglicht eine deutlich schnellere Entwicklung und erlaubt die automatische Generierung von synthesefähigem VHDL-Code auf Basis eines zuvor getesteten Simulink-Modells. Dies erleichtert insbesondere die Erstellung mehrerer Filtervarianten.

Für die Umsetzung eines digitalen IIR-Filters wurde daher ein Simulink-Modell entwickelt, das mithilfe des HDL Coders in VHDL-Code übersetzt wird. Ziel ist es, daraus einen Vivado-kompatiblen IP-Core zu erzeugen, der über eine AXI4-Stream-Schnittstelle in ein FPGA-Design eingebunden werden kann. Dieser modellbasierte Workflow reduziert die Entwicklungszeit und vereinfacht auch die Integration in die bestehende FPGA-Architekturen erheblich. In der Vivado Design Suite von AMD/Xilinx spielen IP-Cores (Intellectual Property Cores) eine zentrale Rolle. Dabei handelt es sich um wiederverwendbare Hardwaremodule, die sich modular in digitale Systeme einfügen lassen. Für die Implementierung von digitalen Filtern bieten IP-Cores einen Vorteil. Die gesamte Filterlogik kann als in sich geschlossenes Modul gekapselt werden. Dadurch wird die Funktionalität des Filters klar abgegrenzt, was sowohl die Wartung als auch die Wiederverwendbarkeit verbessert. Ein einmal entwickelte Filter-IP-Core kann unkompliziert in andere Designs oder Projekte übernommen und betrieben werden, ohne den zugrunde liegenden HDL-Code erneut anpassen zu müssen.

32.1. Modellierung in Simulink

Für die Umsetzung in Simulink wurde der in der DSP HDL Toolbox enthaltene Biquadratic IIR (SOS) filter verwendet. Der Vorteil dieses Blocks gegenüber einer direkten Umsetzung der transponierten Direct Form II liegt in seiner Optimierung für die HDL-Codegenerierung. Er enthält bereits eine integrierte Steuerungslogik für das valid-Signal im AXI-Stream-Protokoll und verfügt zudem über eine vorimplementierte Pipelining-Struktur. Die Probleme bei der Umsetzung einer Direct-Form-Struktur lagen hauptsächlich darin, dass bei der Bitstream-Generierung in Vivado die Timing-Anforderungen nicht eingehalten werden konnten und die Taktfrequenz des Filters entsprechend reduziert werden musste. Wird die Taktfrequenz zu niedrig gewählt, verlängert sich die Zeit der Filterung entsprechend. Hinzu kommt, dass ein einzelner Filter in dieser Struktur vergleichsweise viele Ressourcen auf dem FPGA beansprucht. Eine direkte Implementierung der Direct Form II ohne korrektes Pipelining und weiterführende Optimierungen gestaltet sich zudem als äußerst anspruchsvoll. Aus diesem Grund fiel

die Entscheidung zugunsten einer vorgefertigten Lösung aus der DSP HDL Toolbox. Trotzdem musste die Taktfrequenz letztlich auf 50 MHz begrenzt werden, da bei 100 MHz ebenfalls keine vollständige Einhaltung der Timings erreicht werden konnte.

32.1.1. Direct Form II Transponiert mit Pipelining

Der Filter-Block wurde auf Direct Form II Transponiert eingestellt, da diese Struktur, genau wie die klassische Direct Form II, nur eine Verzögerungskette benötigt und somit weniger Register erforderlich sind. Die transponierte Form verschiebt die Verzögerungselemente auf die Rückkopplungspfade, wodurch die kritische Pfadlänge verkürzt wird. Gleichzeitig bietet sie Vorteile in Bezug auf die numerische Stabilität und ist robuster gegenüber Quantisierungseffekten. Ergänzend ist dieser Filterblock schon gepipelined. Beim *Pipelining* werden innerhalb der Struktur gezielt Register in die kombinatorischen Signalpfade eingefügt. Dadurch werden lange Logikpfade unterbrochen, was die kritische Pfadlänge reduziert. Werte können häufiger zwischengespeichert werden, sodass pro Taktzyklus weniger Rechenoperationen durchgeführt werden müssen. Dies ermöglicht eine höhere maximale Taktfrequenz und verbessert insgesamt die Timing-Eigenschaften der Schaltung. Rückkopplungswege, die bei IIR-Filtern eine Herausforderung darstellen, können so auch bei hoher Verarbeitungsgeschwindigkeit stabil betrieben werden.

32.1.2. Festkommaarithmetik

Die gesamte Filterimplementierung verwendet Festkommaarithmetik, da FPGAs standardmäßig keine native Gleitkomma-Hardware besitzen. In digitalen Systemen ist die Verwendung von Festkomma eine zentrale Voraussetzung, da alle Signale und Rechenergebnisse nur mit einer endlichen Wortbreite dargestellt und verarbeitet werden können. Dies führt zwangsläufig zu Rundungs- und Quantisierungsfehlern, die insbesondere bei rekursiven Systemen wie IIR-Filtern zu Stabilitätsproblemen führen können, wenn sie nicht sorgfältig berücksichtigt werden. Ein Aspekt der Festkommaarithmetik ist die Wahl einer geeigneten Zahlendarstellung. Um die begrenzte Bitbreite optimal zu nutzen, ist eine sorgfältige Skalierung notwendig. Sie verhindert Überläufe und stellt sicher, dass die numerische Auflösung bestmöglich ausgeschöpft wird. Außerdem wirkt sich die gewählte Wortlänge auf den kritischen Pfad im digitalen System aus. Je länger die arithmetischen Operationen, desto größer sind die resultierenden Verzögerungszeiten, was wiederum die maximal erreichbare Taktfrequenz limitiert.

Angesichts dessen, wird für alle Signalpfade ein vorzeichenbehafteter 32-Bit-Festkommatyp mit 16 Nachkommabits verwendet. Dieses Format wurde sowohl in Simulink als auch in der späteren Hardwareimplementierung experimentell validiert und hat sich dabei als optimale Wahl erwiesen.

32.1.3. AXI4-Stream-Schnittstelle

Die Integration der Filter in das FPGA-System erfolgt über eine *AXI4-Stream-Schnittstelle*, einem Teil der standardisierten AMBA AXI4-Interface-Protokollfamilie. Diese Schnittstellenarchitektur stammt ursprünglich von Arm und wird von AMD/Xilinx weitläufig in Vivado-Designs eingesetzt.

Die AXI4-Familie setzt sich dabei zusammen aus *AXI4* für adressierte Hochgeschwindigkeits-Datenübertragungen mit Burst-Unterstützung, *AXI4-Lite* für einfache Steuerregister-Kommunikation ohne Burst-Funktionalität und *AXI4-Stream* für kontinuierliche, nicht-adressierte Datenströme. Letzteres erlaub mehrere Datenströme mit unterschiedlichen Datenbreiten über denselben Interconnect zu übertragen. Die Kommunikation basiert auf einem einfachen Handshake-Mechanismus mit den Signalen TVALID und TREADY. Ein Transfer erfolgt nur, wenn beide Signale gleichzeitig aktiv sind. Der Sender (Master) setzt TVALID sobald gültige Daten vorliegen, und der Empfänger (Slave) signalisiert mit TREADY, dass dieser bereit ist Daten zu empfangen. Der Sender darf *nicht auf TREADY warten*, bevor er TVALID aktiviert. Der Sender *muss TVALID so lange halten*, bis der Handshake abgeschlossen ist. Der Empfänger darf TREADY auch vor TVALID setzen, muss er aber nicht. Neben den verpflichtenden Signalen TVALID, TREADY und TDATA existieren im *AXI4-Stream-Protokoll* auch optionale Signale wie TID und TLAST. Das Signal TID (Transaction ID) ermöglicht es, mehrere logische Datenströme innerhalb eines gemeinsamen physischen AXI4-Stream-Kanals voneinander zu unterscheiden. Dies ist besonders relevant, wenn verschiedene Datenquellen über denselben Stream multiplexed werden. TLAST kennzeichnet bei paketbasierter Übertragung das Ende eines Datenpakets. Es wird gesetzt, sobald das letzte Datenwort eines Transfers übertragen wird. Die Verwendung von AXI4-Stream in Vivado hat den großen Vorteil, dass IP-Blöcke auf standardisierte Weise verbunden werden können. Über *AXI-Interconnects* oder *AXI SmartConnect*-Komponenten kann der Datenfluss zwischen IPs automatisch geroutet werden, was die Einbindung vereinfacht.

32.1.4. Modell und Aufbau

Die Numerator- und Denominator-Koeffizienten des Filters werden direkt aus dem MATLAB-Workspace übernommen und stammen aus der vorberechneten SOS-Matrix. Der Gain-Faktor wird in die Numerator-Koeffizienten eingerechnet, um eine stabile Skalierung zu gewährleisten. Dadurch wird verhindert, dass während der Berechnung Zwischenwerte entstehen, die außerhalb des darstellbaren Wertebereichs liegen.

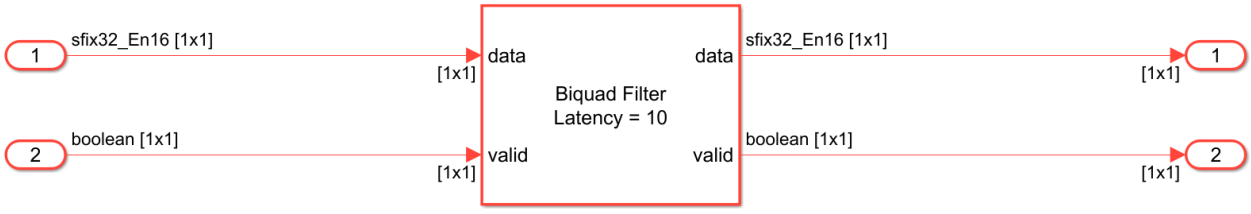
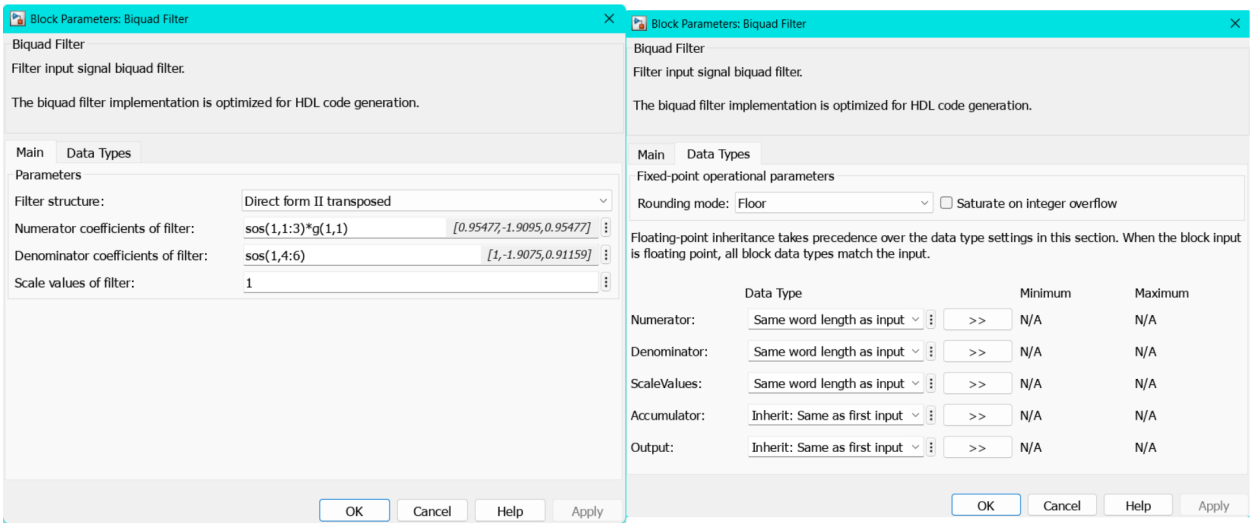


Abbildung 32.1.: Simulink Filter: Subsystem

Der Filter ist in ein separates Subsystem eingebettet, um die eigentliche Filterlogik klar von zusätzlichen Test- und Steuerungselementen zu trennen. Um den Datentyp durchgängig korrekt zu setzen, wird ein Data Type Conversion-Block vor dem Filter-Subsystem eingefügt. Der Filter selbst

ist so eingestellt, dass die Koeffizienten den Datentyp des Eingangssignals übernehmen. Die genauen Einstellungen sehen wie Folgt aus:



(a) Biquad Filter: Main

(b) Biquad Filter: Datatypes

Abbildung 32.2.: Biquad Filter: Settings

Ergänzend enthält das Modell Komponenten zur Test- und Signalanalyse. Dazu gehören Elemente, die Testsignale aus dem Workspace einlesen, ein gültiges AXI-Stream Valid-Signal zur Simulation erzeugen, sowie Blöcke zum Zurückschreiben der Ergebnisse ins MATLAB-Workspace.

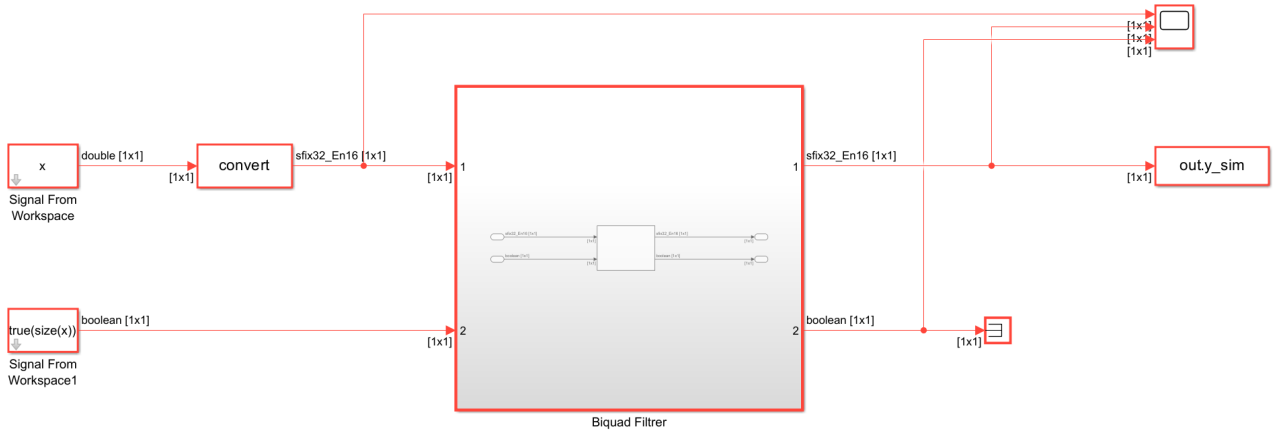


Abbildung 32.3.: Simulink Filter

32.2. HDL-Coder und Workflow Advisor

Für die Überführung des Filtermodells in eine hardwaretaugliche Form kommt der *HDL Coder* von MathWorks zum Einsatz. Dieses Tool ermöglicht die automatische Generierung von synthesefähigem HDL-Code aus einem Simulink-Modell oder MATLAB-Algorithmus. Dadurch entfällt die manuelle Erstellung von VHDL- oder Verilog-Code. Der HDL Coder unterstützt einen modellbasierten Entwicklungsablauf, bei dem speziell für die Hardwaregenerierung ausgelegte, HDL-kompatible Simulink-Blöcke verwendet werden. Diese gewährleisten eine korrekte Abbildung der Signalverarbeitung im späteren FPGA-Design. Die automatisierte Codegenerierung erfolgt über den HDL Workflow Advisor, der den Nutzer schrittweise durch den gesamten Prozess führt. Abhängig von der gewählten Zielplattform wird dabei festgelegt, welche zusätzlichen Schnittstellen generiert werden sollen. So kann ein IP-Core mit AXI4-Stream- oder AXI4-Lite-Schnittstellen erzeugt werden. Die Ein- und Ausgänge des Simulink-Modells werden entsprechend den HDL-I/O-Ports oder AXI-Schnittstellen zugewiesen. Nach Abschluss der Konfiguration generiert der HDL Coder automatisch den vollständigen HDL-Code, einschließlich aller erforderlichen Constraints-Dateien (XDC) sowie optionaler Testbenches. Der gesamte Ablauf wird strukturiert durchlaufen und dokumentiert. Im Rahmen dieser Implementierung wird auf Basis der Codegenerierung ein Vivado-kompatibler IP-Core erzeugt, der in ein lokales IP-Repository abgelegt wird. Dieser IP-Core kann anschließend ohne zusätzliche Anpassungen in ein Vivado Block Design eingebunden werden. Der kombinierte Einsatz von HDL Coder und HDL Workflow Advisor ist besonders dann vorteilhaft, wenn begrenzte Erfahrung in der manuellen HDL-Entwicklung vorliegt oder eine zügige, reproduzierbare Prototypenerstellung erforderlich ist. Die automatisierte Vorgehensweise reduziert Fehlerquellen, verbessert die Wiederverwendbarkeit und gewährleistet, dass alle Schnittstellenanforderungen bereits im Modellierungsprozess berücksichtigt werden. So konnten die IIR-Filter mit AXI4-Stream-Anbindung hardwaregerecht umgesetzt werden.

32.2.1. Modellvorbereitung für den HDL-Coder

Damit das Simulink-Modell für den HDL-Coder verwendet werden kann, müssen einige Einstellungen in Simulink selbst durchgeführt werden. Dieses Setupt lässt sich im *Comand Window* in Matlab automatisch gestalten, mit der Voraussetzung, dass Vivado installiert ist. Für dieses Projekt wird Vivado 2022.1 verwendet, dies ist die aktuellste Version von Vivado welche von Pynq unterstützt wird.

Das Setup in Simulink lässt sich mit der Funktion *hdlsetup(modelname)* ausführen. Diese Funktion konfiguriert das geöffnete Modell für den HDL-Workflow, indem sie notwendige Pfade und Einstellungen für den HDL Coder ergänzt. Sie bereitet Simulink gezielt für die Codegenerierung und FPGA-Implementierung vor. Die Funktion muss pro Modell nur einmalig ausgeführt werden. Nach erfolgreicher Ausführung wird das Simulink-Modell typischerweise durch einen roten Rahmen markiert, ein Hinweis darauf, dass es für die HDL-Codegenerierung vorbereitet ist.

Die Funktion *hdlsetuptoolpath(path)* konfiguriert den Pfad zu externen Synthese-Tools, die für Codegenerierung benötigt werden. Dieser Schritt ist erforderlich, damit die HDL Workflow Advisor-Umgebung später auf die notwendigen Werkzeuge zugreifen und einen IP-Core generieren kann. Der Befehl sollte vor dem Öffnen des HDL Workflow Advisor ausgeführt werden, andernfalls kann

der Toolpfad nicht korrekt übernommen werden. Es empfiehlt sich daher, diesen Befehl in ein MATLAB-Skript zu integrieren.

32.2.2. Einstellungen vom HDL Workflow Advisor

Im Folgenden werden die wichtigsten Einstellungen beschrieben, die für die Erstellung und Konfiguration des IP-Cores vorgenommen wurden.

32.2.2.1. 1.1 Set Target Device and Synthesis Tool

Im ersten Schritt werden im HDL Workflow Advisor die grundlegenden Parameter für die Zielhardware festgelegt. Als **Target Workflow** wird *IP Core Generation* ausgewählt, wodurch der Workflow Advisor für die Erstellung eines eigenständigen IP-Cores konfiguriert wird. Die **Target Platform** ist auf *Generic Xilinx Platform* eingestellt, es wird also keine board-spezifische Plattform gewählt, sondern eine generische Xilinx-Zielumgebung genutzt. Als **Synthesis Tool** kommt *Xilinx Vivado* in der Version 2022.1 zum Einsatz. Damit wird definiert, dass die Synthese, die Implementierung und die spätere IP-Integration in einem Vivado-Projekt erfolgen. Unter **Family**, **Device**, **Package** und **Speed** sind die Werte *Zynq*, *xc7z020*, *clg400* und *-1* gesetzt. Diese Einstellungen entsprechen den Anforderungen des PYNQ-Z2 Boards. Die Werte für **Package** und **Speed** wurden dabei automatisch erkannt und übernommen. Anschließend wird der Zielpfad angegeben, an dem die generierte IP gespeichert werden soll. Auf diese Weise ist sichergestellt, dass der IP-Core später direkt in Vivado eingebunden werden kann.

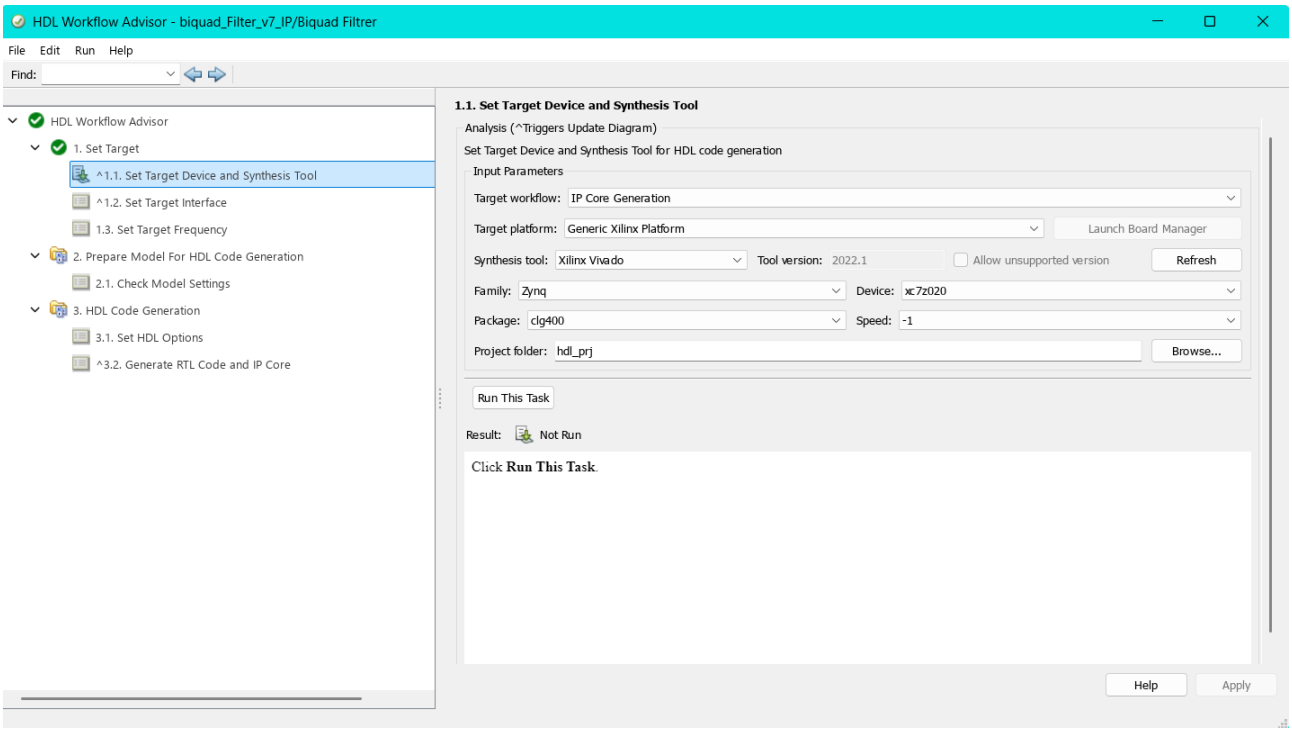


Abbildung 32.4.: HDL Workflow: Target Device

32.2.2.2. 1.2 Set Target Interface

In diesem Schritt werden die Ein- und Ausgänge des Simulink-Modells den Schnittstellen der AXI-Stream-Plattform zugewiesen. Die Option **Processor/FPGA Synchronization** ist auf *Free running* gesetzt, da die spätere Anwendung als kontinuierlicher Datenstrom (Streaming) ausgelegt ist. Die Übersicht in der Tabelle bietet zusätzlich einen guten Überblick über die zugewiesenen Datentypen der Ports. Bei den Signalen *In1* und *Out1* ist erkennbar, dass der zuvor über einen *Data Type Conversion-Block* eingestellte Festkommadatentyp übernommen wurde. Wichtig ist außerdem, dass die Ein- und Ausgänge für das *Valid-Signal* (*In2* und *Out2*) den Datentyp *boolean* besitzen, da dies der AXI-Stream-Konvention entspricht. Für den Ausgang *Out1* kann über die Schaltfläche **Options** zusätzlich die Größe des Ausgangspuffers konfiguriert werden. Standardmäßig ist dieser auf *1024* Bits eingestellt. Die Größe des Ausgangspuffers wurde auf 2^{20} eingestellt. Diese definierte Ausgangspuffergröße bestimmt die maximale Datenmenge, die pro Übertragung an den Filter gesendet werden kann. Diese Größe darf später weder unter- noch überschritten werden. Wird der Ausgangspuffer zu klein gewählt, muss das Eingangssignal in viele einzelne Übertragungen aufgeteilt werden. Da sich der IIR-Filter bei jeder Übertragung erneut einschwingt und keine Zustände zwischen den Übertragungen erhalten bleiben, kann es zu hörbaren Beeinträchtigungen kommen. Dieses wiederholte Einschwingen verstärkt Hintergrundrauschen und numerische Störungen, was sich in schwankender Lautstärke und zunehmendem Rauschen bemerkbar macht.

Der Haken bei **Generate default AXI4 slave interface** sollte entfernt werden, sofern er gesetzt ist. Andernfalls wird automatisch eine AXI4-Lite-Schnittstelle erzeugt. Diese wird üblicherweise

für die Registersteuerung verwendet. Da die hier entwickelte IP jedoch für einen kontinuierlichen Datenstrom ausgelegt ist, ist eine Steuer-Schnittstelle nicht erforderlich. Sie würde das Design unnötig verkomplizieren.

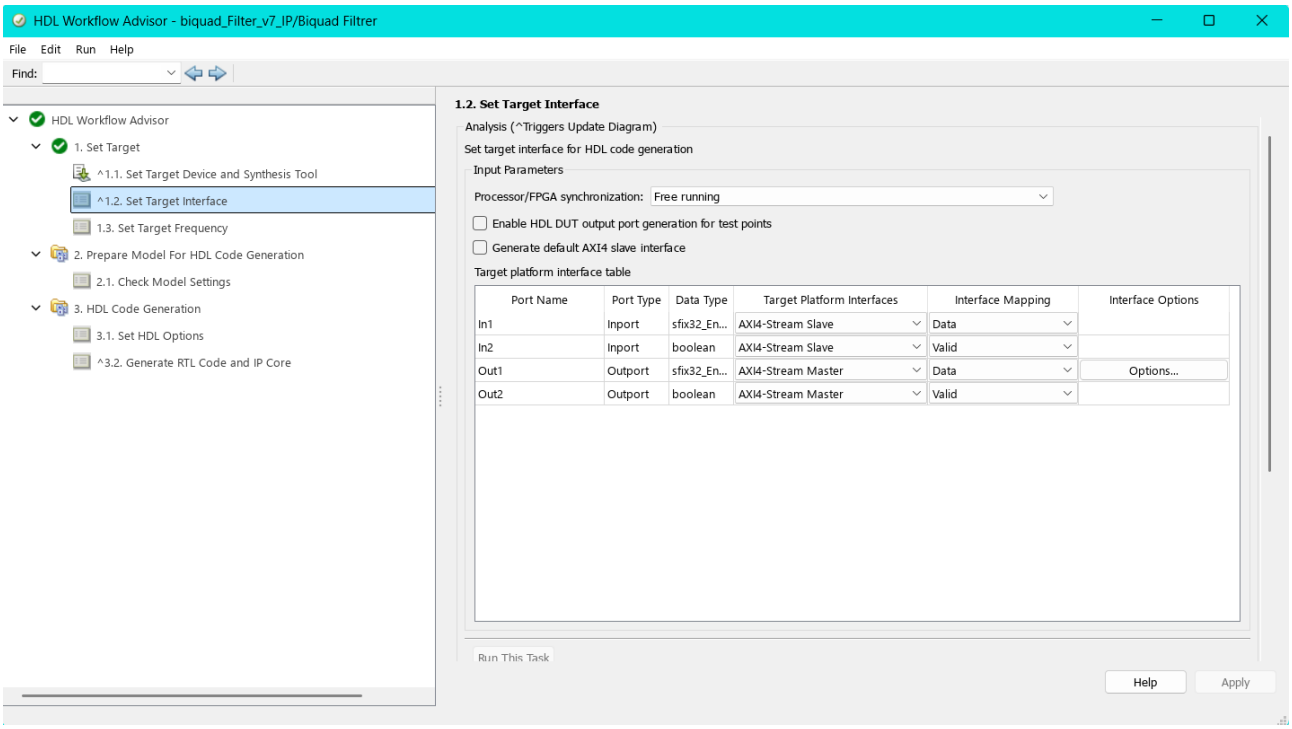


Abbildung 32.5.: HDL Workflow: Target Interface

32.2.2.3. 1.3 Set Target Frequency

An dieser Stelle wird die Zielfrequenz der zu generierenden IP festgelegt. Ursprünglich war diese auf 100 MHz gesetzt. Da jedoch bei der späteren Bitstream-Erzeugung die Timing-Anforderungen nicht erfüllt werden konnten, wurde die Frequenz auf 50 MHz halbiert. Diese reduzierte Taktfrequenz zeigte im Design stabile Timing-Ergebnisse und wurde daher für die weitere Implementierung beibehalten.

32.2.2.4. 2.1 Check Model Settings

In diesem Schritt werden die verwendeten Simulink-Blöcke daraufhin überprüft, ob sie für die HDL-Codegenerierung geeignet sind. Zusätzlich kann ein **Industry Standard Check** durchgeführt werden, der das Modell hinsichtlich Industriestandards bewertet. Dabei werden häufig Warnungen ausgegeben, insbesondere bezüglich der Namenskonventionen, wenn diese nicht den typischen Industriestandards entsprechen. Für die hier verfolgte Anwendung ist es nicht erforderlich, den Industry Standard Check zu aktivieren, und es wird empfohlen, diesen nicht auszuführen, um unnötige Warnmeldungen zu vermeiden.

32.2.2.5. 3.1 Set HDL Options

Hier werden die Einstellungen für die Codegenerierung getroffen. Im Bereich **Optimization/General** wurden die Einstellungen größtenteils unverändert übernommen. Dabei ist es jedoch wichtig darauf zu achten, dass die Option *Enable-based constraints* deaktiviert ist. Diese Funktion erzeugt zusätzliche Timing-Beschränkungen auf Basis von *Enable-Signalen*, was insbesondere bei einfacheren Designs oder kontinuierlich laufenden IPs zu unerwünschten Timing-Problemen führen kann.

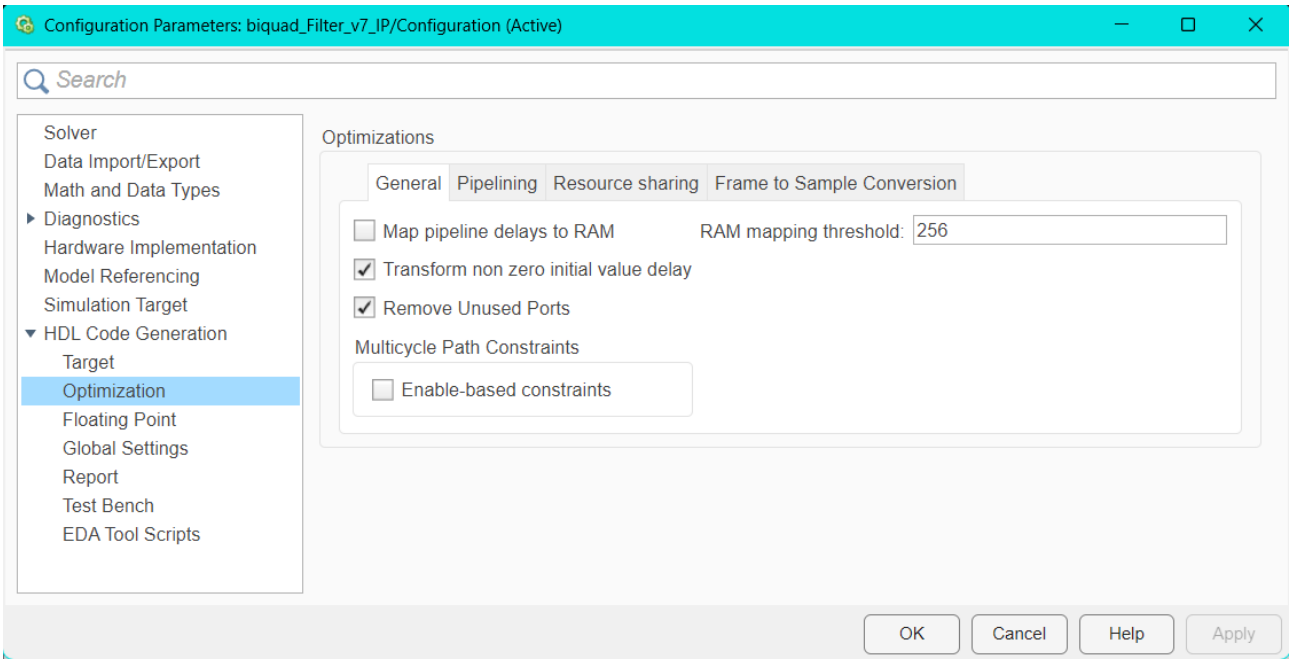


Abbildung 32.6.: HDL Workflow: HDL Options/ Optimization/ General

Im Abschnitt **Optimization/Pipelining** blieben die Voreinstellungen ebenfalls weitgehend bestehen. Die Option *Adaptive Pipelining* kann dabei aktiviert werden. Diese Einstellung ermöglicht normalerweise eine automatische Optimierung der Pipeline-Struktur im Datenpfad. Da jedoch ein fertiger Block aus der DSP HDL Toolbox verwendet wurde, der intern bereits gepipelined ist, kann diese Option in diesem Fall auch deaktiviert bleiben. Während der Entwicklung wurde Adaptive Pipelining zunächst genutzt, um eine direkte Nachbildung der Direct-Form-Struktur zu pipelinen. Diese automatische Pipelining-Methode reichte jedoch nicht aus, um das Modell ausreichend zu optimieren.

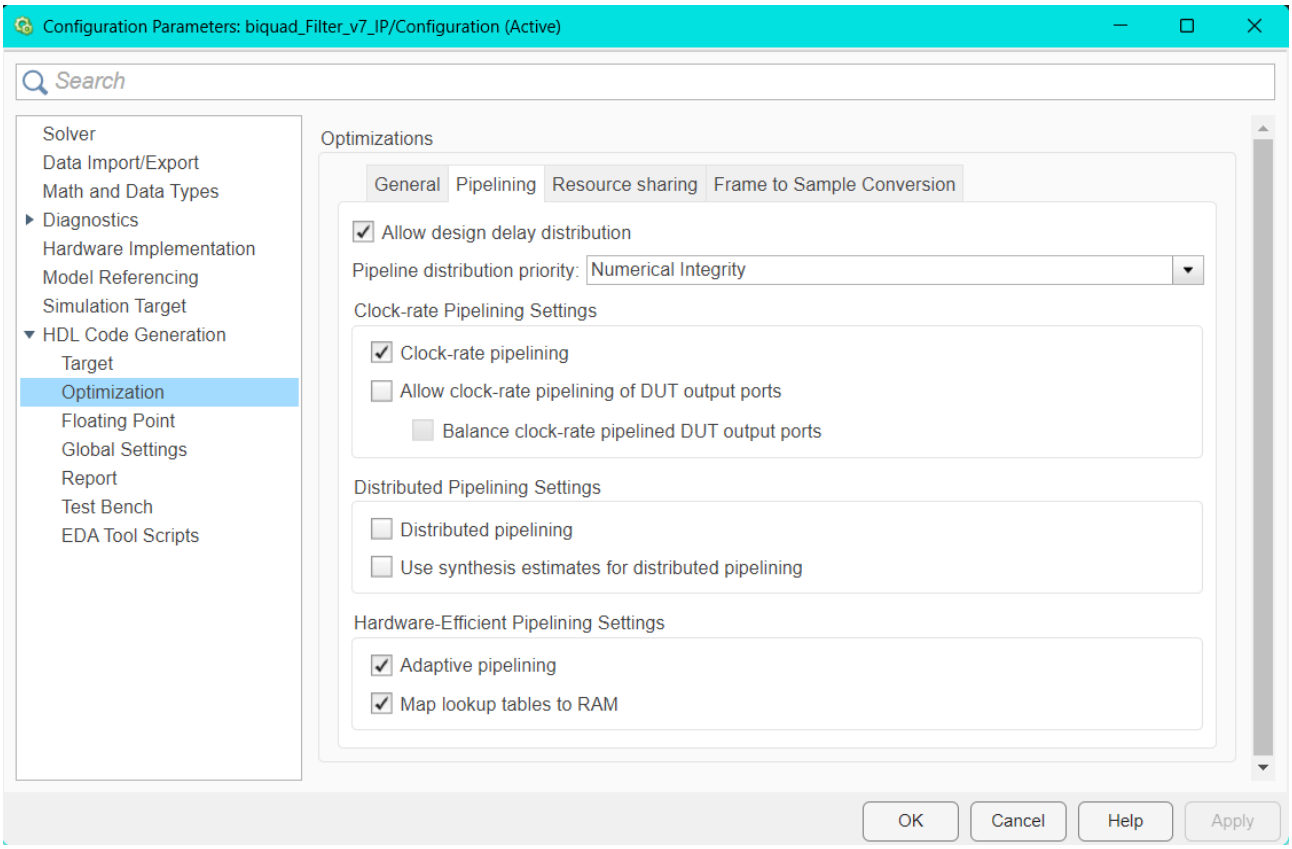


Abbildung 32.7.: HDL Workflow: HDL Options/ Optimization/ Pipelining

Die **Global Settings** wurden größtenteils automatisch gesetzt. Dennoch sollte unbedingt geprüft werden, dass der *Reset type* auf *Synchronous* und der *Reset asserted level* auf *Active-high* eingestellt ist. Eine inkorrekte Reset-Konfiguration kann andernfalls beim Generieren der IP zu Warnungen oder Fehlern im Abschlussbericht führen.

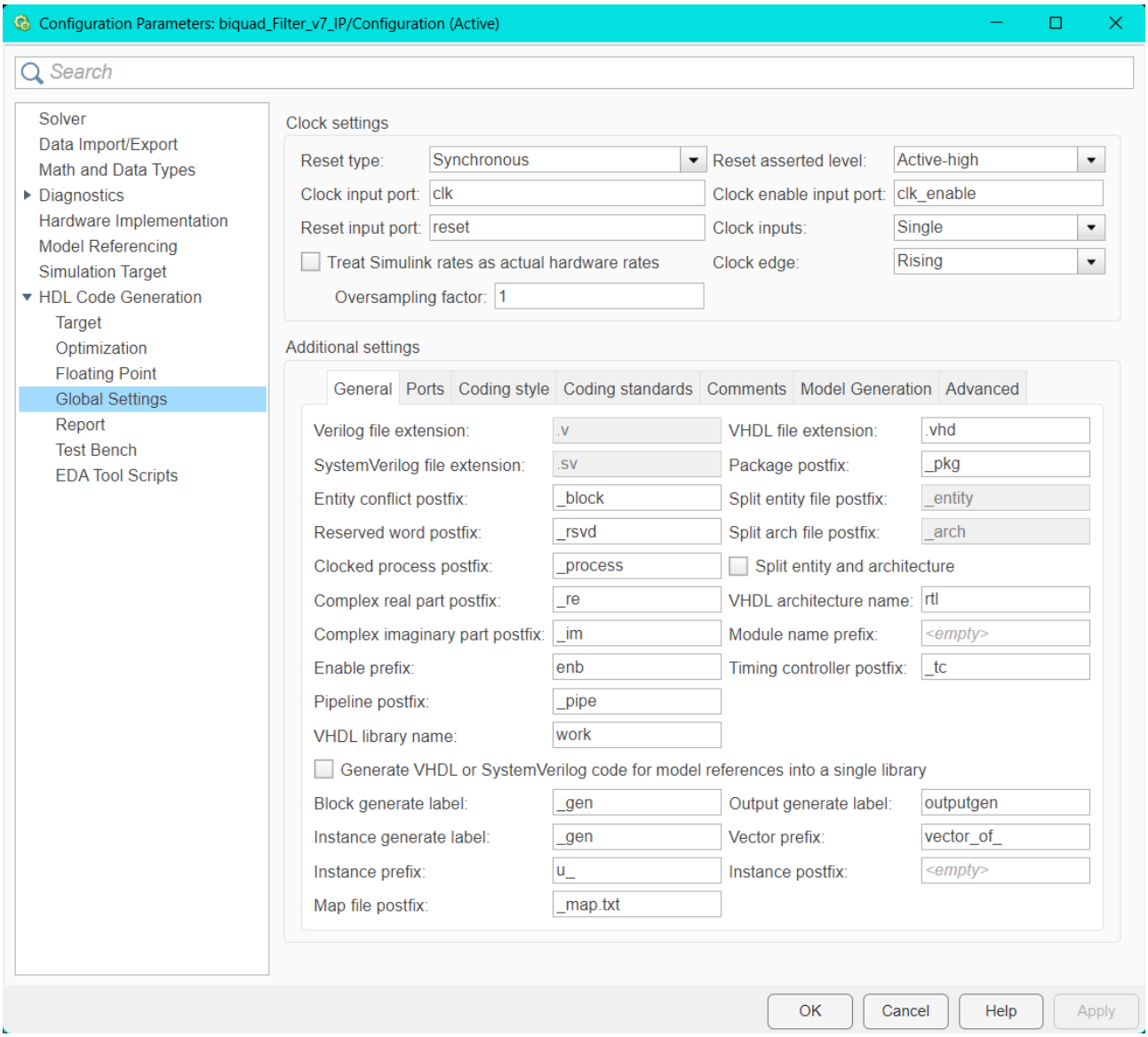


Abbildung 32.8.: HDL Workflow: HDL Options/ Global Settings

32.2.2.6. 3.2 Generate RTL Code and IP Core

Hier lässt sich der zu generierende IP-Core benennen und eine Versionsnummer vergeben. Für die IP sollte ein aussagekräftiger Name gewählt werden, damit sie später in Vivado leicht auffindbar ist.

Um die IP zu generieren und alle vorgenommenen Einstellungen anzuwenden und zu überprüfen, klickt man mit der rechten Maustaste auf den Schritt **3.2 Generate RTL Code and IP Core** und wählt *Run to Selected Task* aus. Dabei werden alle vorangehenden Schritte automatisch durchlaufen. Sollte

ein Fehler auftreten oder eine Prüfung fehlschlagen, wird der Vorgang an der entsprechenden Stelle abgebrochen.

Nach der Generierung wird der IP-Core erstellt und ein Bericht geöffnet. Dieser enthält sowohl mögliche Warnungen als auch den erzeugten HDL-Code. Zusätzlich bietet der vollständige Report einen Überblick über die Ressourcennutzung, darunter beispielsweise die Anzahl verwendeter LUTs, Flip-Flops oder Multiplikatoren.

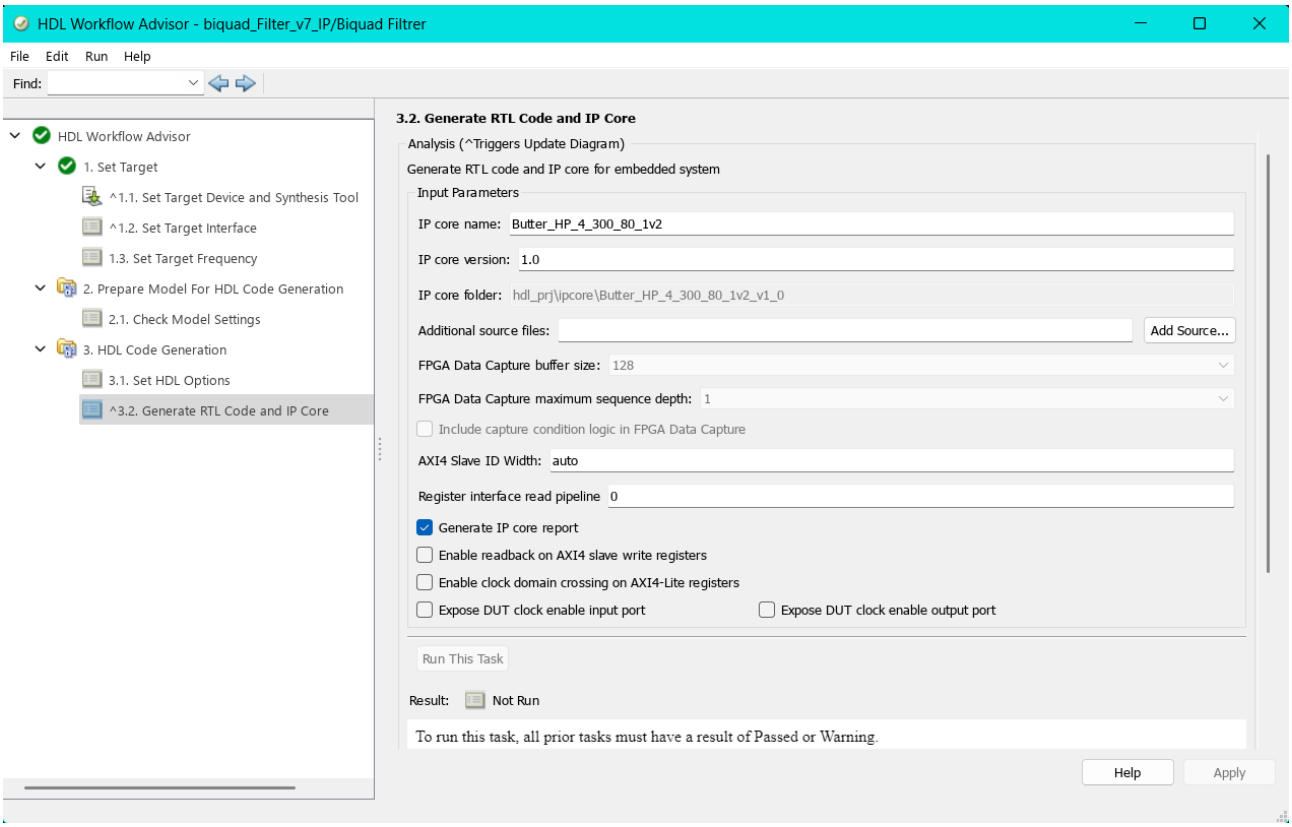


Abbildung 32.9.: HDL Workflow: Generate RTL Code and IP Core

Für jeden Filter wird eine eigene IP generiert. Dabei wird jeweils dasselbe Simulink-Modell verwendet, lediglich die Koeffizienten werden angepasst.

32.3. Design in Vivado

Für die Aufnahme und Wiedergabe von Audio über das Pynq-Board wurde prinzipiell der Audioanteil aus dem Base-Overlay übernommen und die nicht benötigten Elemente entfernt. Die Einstellungen der Blöcke wurden ebenfalls übernommen. Die Constraints wurden so angepasst, dass die Audiofunktionalität erhalten blieb. Die Constraints weisen unter anderem den Benötigten internen PINs Namen zu, damit diese besser nachvollzogen werden können. Es ist notwendig, dass die

Namen an dem Ein- und Ausgängen aus den Constraints übernommen werden. Für den Betrieb des Audio-Codecs auf dem Pynq-Z2 sind spezielle IP-Cores von Pynq notwendig. Diese IPs sowie auch die Constraints befinden sich im PYNQ GitHub Repository und müssen in das Projekt eingebunden werden. Vorgefertigte sowie auch eigens erstellte IP-Cores lassen sich in Vivado unter *Project Manager/Settings/IP/Repository* in das Projekt einbinden.

Teile des Designs orientieren sich an den offiziellen Tutorials von **Cathal McCabe** , die im AMD PYNQ-Forum verfügbar sind. Besonders hilfreich waren die Anleitungen Creating a new hardware design for PYNQ und PYNQ DMA. Teile dieser Tutorials, insbesondere grundlegende Strukturkomponenten, wurden angepasst übernommen.

32.3.1. Audio Codec Controller

Der *audio_codec_controller* ist eine eigene IP von PYNQ und erzeugt die vom Audio-Codec benötigten Steuersignale sowie den I^2S Datenstrom. Während im ursprünglichen Overlay ein *Clocking Wizard* eingesetzt wird, um aus dem 100 MHz-Systemtakt ein 10 MHz-Taktsignal zu erzeugen, stellt die überarbeitete Variante dieses 10 MHz-Signal direkt aus dem Processing System (PS) bereit, inklusive eines synchronisierten Reset-Signals. Diese Vorgehensweise hat den Vorteil, dass keine zusätzlichen Clock-Domain-Warnungen im Vivado auftreten.

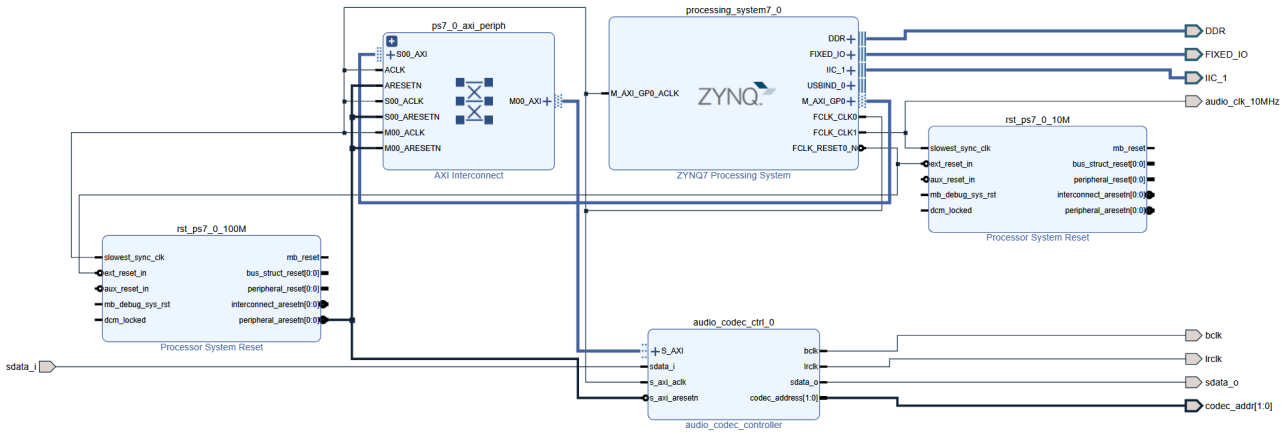


Abbildung 32.10.: Reference Design

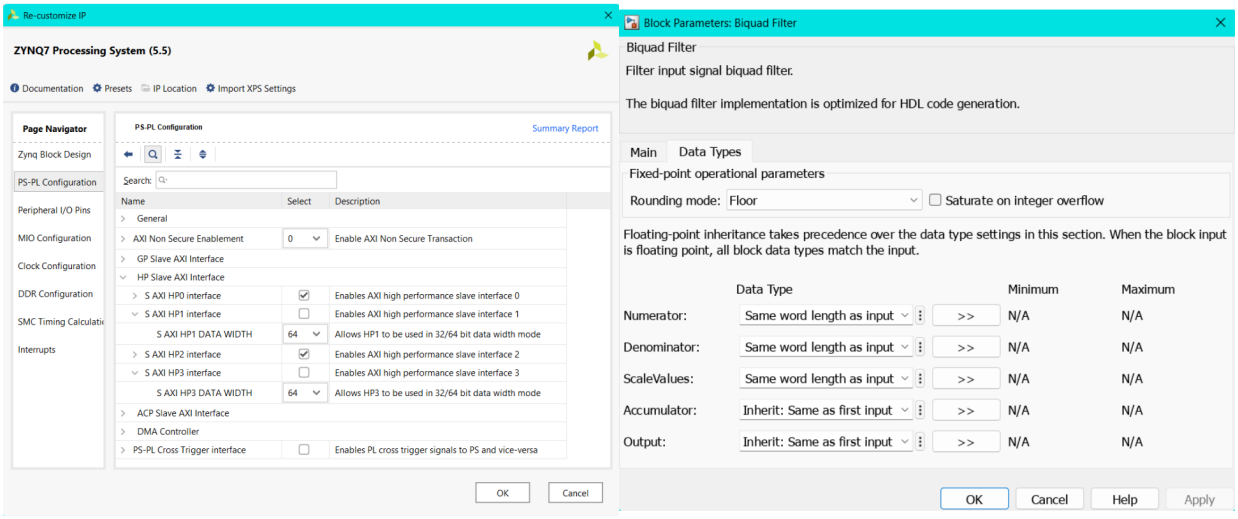
32.3.2. Processing System (PS)

Da das Design später über ein Jupyter Notebook gesteuert werden soll, wie es für die PYNQ-Plattform typisch ist, reicht eine reine RTL-Implementierung innerhalb der programmierbaren Logik nicht aus. Stattdessen wird das ZYNQ Processing System benötigt, das die Schnittstelle zwischen der Software-Ebene (Python/Jupyter) und der Hardware-Ebene (PL) bildet. Über das Processing System (PS) werden alle zentralen Taktsignale für das Design bereitgestellt. Zudem übernimmt das PS die Steuerung des Audio-Codecs sowie des AXI Direct Memory Access (AXI-DMA), über den die Filter mit dem PS verbunden sind. Die AXI-DMA-Verbindung ermöglicht einen Datenaustausch zwischen

dem Arbeitsspeicher des PS und den in der programmierbaren Logik (PL) implementierten Filtern. Auf diese Weise kann das Jupyter Notebook über das PS Daten an die Filter senden und deren Ausgaben empfangen. Damit der PS die korrekten Takt- und Schnittstellensignale erzeugt, muss er im Vivado-Design entsprechend konfiguriert werden.

32.3.2.1. PS-PL Cinfiguration

Um Daten über die AXI-DMA-Schnittstellen an die Filter übergeben zu können, werden zusätzliche Anschlüsse am Processing System (PS) benötigt. Der PS besitzt intern zwei physische Zugänge zum DRAM, wobei sich jeweils zwei Ports (HP0/1 und HP2/3) einen gemeinsamen Zugriff teilen. Für dieses Design werden lediglich zwei High-Performance-Ports benötigt, weshalb HP0 und HP2 aktiviert werden. Die Datenbreite (Data Width) kann auf dem Standardwert von 64 Bit belassen werden.



(a) Vivado:PS Settings/ PS-PL-Configuration (b) Vivado:PS Settings/ Clock Configuration

Abbildung 32.11.: Vivado:PS Settings

32.3.2.2. Clock Configuration

Alle benötigten Takt-Signale werden direkt im Processing System (PS) generiert und den jeweiligen Komponenten zugewiesen. Der Audio-Codec benötigt ein 10 MHz Taktsignal, während der Audio Codec Controller ein 100 MHz Taktsignal erfordert. Andere Frequenzen werden von dieser IP nicht unterstützt, was in Vivado durch eine entsprechende Warnung angezeigt wird. Die Filter-IP-Cores wurden auf 50 MHz ausgelegt und benötigen daher ebenfalls ein passendes 50 MHz-Taktsignal.

32.3.2.3. Peripheral I/O Pins

Für die Kommunikation mit dem Audio Codec Controller wird eine I^2C-Verbindung benötigt. Diese Schnittstelle lässt sich in der PS-Konfiguration unter *Peripheral I/O Pins* aktivieren, indem der entsprechende Haken gesetzt wird. In diesem Fall wurde *I2C 1* ausgewählt.

32.3.3. AXI Direct Memory Access

Die Anbindung der Filter-IP-Cores an das Processing System (PS) erfolgt über einen AXI Direct Memory Access (AXI-DMA). Dieser ermöglicht Daten direkt zwischen dem Speicher des Processing Systems (PS) und der programmierbaren Logik (PL) zu übertragen, ohne dass der Prozessor selbst aktiv Daten verschieben muss. Da im Design vier Filter parallel verwendet werden, werden auch vier AXI-DMA-Instanzen benötigt, die alle identisch konfiguriert werden. Eine konsistente Namenskonvention ist hierbei praktisch, da im Jupyter Notebook später über diese Namen auf die jeweiligen DMA-Schnittstellen zugegriffen wird.

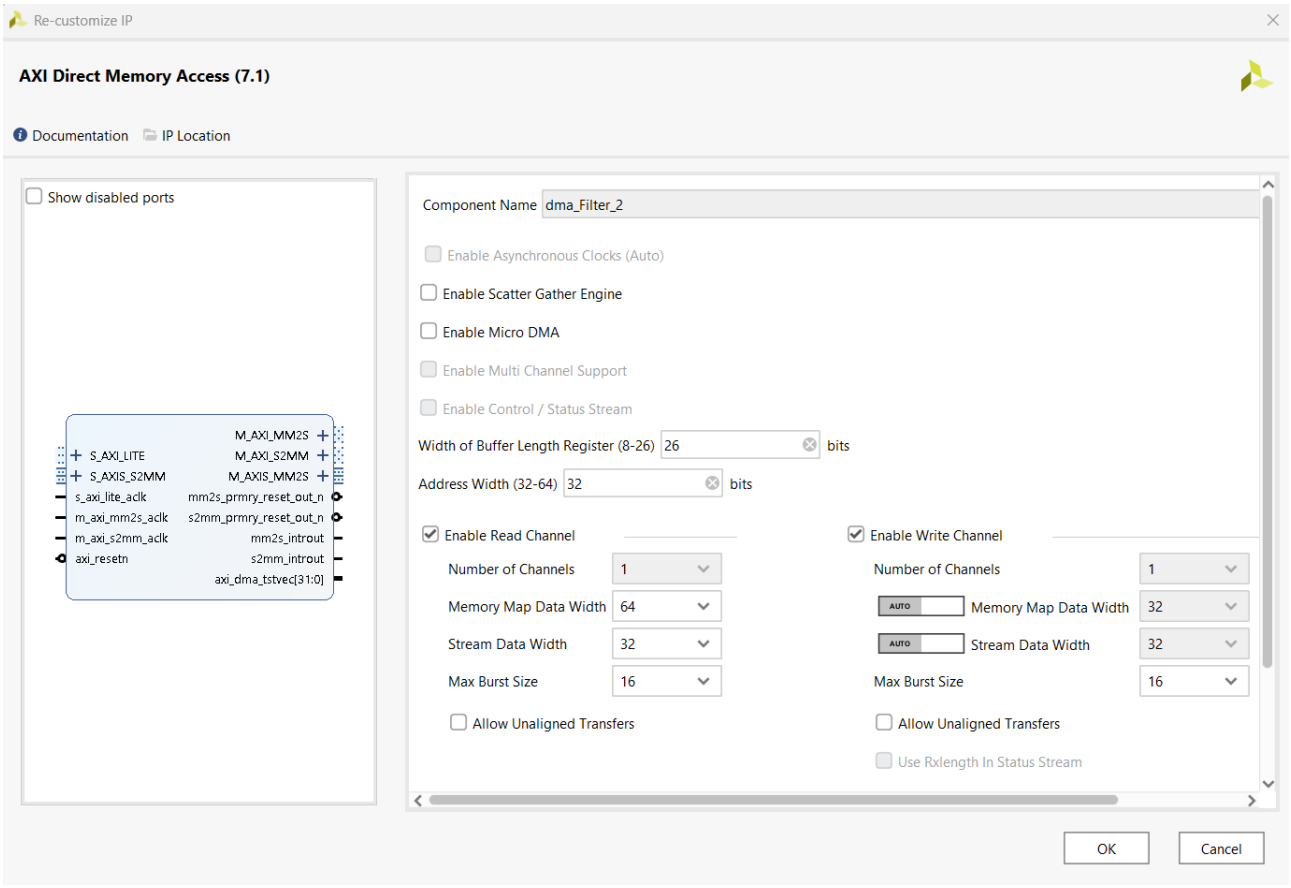


Abbildung 32.12.: AXI-DMA Config

Bei der Konfiguration ist besonders darauf zu achten, dass sowohl *Enable Scatter Gather Engine* als auch *Enable Micro DMA* deaktiviert werden, da diese Funktionen in diesem Design nicht benötigt werden. *Allow unaligned transfers* sollte ebenfalls deaktiviert bleiben, da diese Funktion auf dem Pynq nicht unterstützt wird. Die *Width of Buffer Length Register* wird auf den maximalen Wert von 26 gesetzt. Auch wenn dieser Wert die Anforderungen der Filter übersteigt, wurde er aus Gründen der Robustheit und zur Vermeidung zukünftiger Einschränkungen bewusst beibehalten. Ein großer Puffer verhindert zudem, dass Fehler fälschlicherweise dem DMA zugeordnet werden, obwohl sie tatsächlich im Filter auftreten.

Die *Address Width* bleibt auf dem Standardwert von 32 Bit. Die *Stream Data Width* wird entsprechend der Konfiguration der *HP0/HP2-Ports* des PS auf 64 Bit gesetzt. Auch hier gilt Lieber großzügig dimensionieren als zu knapp, was ebenfalls aus der Entwicklungsphase übernommen wurde.

Die *Max Burst Size* wird auf 16 Bit gesetzt, obwohl die Burst-Funktion im eigentlichen Betrieb nicht verwendet wird. Für den *Write Channel* wird die Einstellung auf *Auto* belassen, wodurch automatisch die Parameter des Read-Channels übernommen werden.

32.3.4. Verbindungen im Blockdesign

Vivado bietet die Möglichkeit, Blöcke automatisch anhand von vorgeschlagenen Verbindungen miteinander zu verbinden. Bei standardisierten Schnittstellen sind diese Vorschläge in der Regel korrekt und können den Designprozess beschleunigen. Dennoch sollte stets überprüft werden, ob die vorgeschlagene Verbindung tatsächlich der gewünschten entspricht. Bei Verwendung dieser Funktion fügt Vivado oft automatisch unterstützende Blöcke wie beispielsweise einen *AXI Interconnect* hinzu, was zur besseren Übersichtlichkeit und Strukturierung des Designs beiträgt.

Eine Ausnahme bilden jedoch die Ein- und Ausgänge, die über die *Constraints-Datei (XDC)* definiert sind. Für diese Signale funktioniert die automatische Verbindung nur eingeschränkt oder fehlerhaft. Daher wird empfohlen, diese Verbindungen manuell vorzunehmen, um Fehler im späteren Designverlauf zu vermeiden.

32.3.4.1. S_AXI (S_AXI_LITE)

Dabei handelt es sich um die *AXI-Lite-Schnittstelle*, die für die Registersteuerung benötigt wird. Sowohl der *Audio-Codec-Controller* als auch die *AXI-DMA-Module* besitzen jeweils eine solche Verbindung. Diese werden allerdings auf unterschiedliche Clock-Frequenzen eingestellt. Der *Audio Codec Controller* wird auf 100MHz eingestellt und der *Axi-DMA* auf 50MHz.

32.3.4.2. High performace slave interface

Dies sind die Verbindungen an den HP0/2-Ports des Processing Systems (PS), über die die Daten zwischen dem BRAM und dem AXI-DMA übertragen werden. Diese Anschlüsse werden ausschließlich für die AXI-DMA-Kommunikation genutzt. Der *M_AXI_MM2S*-Port ist der *Read Channel*, und *M_AXI_S2MM* ist der *Write Channel*. Da die AXI-DMA-Module als Master fungieren, sind sie in

der Lage, sowohl Lese- als auch Schreibzugriffe durchzuführen. Der PS übernimmt hierbei die Slave-Rolle. Diese Verbindungen werden bei allen vier AXI-DMA-Instanzen auf die gleiche Weise hergestellt. An den *HP0* wird der *M_AXI_MM2S* angeschlossen und an den *HP2* der *M_AXI_2SMM*, beide auf 50MHz.

32.3.4.3. AXI-Stream

Die Filter sowie die AXI-DMA-Module besitzen jeweils einen *S_AXIS_2SMM* bzw. *AXI4_Stream-Slave* (Slave) und einen *M_AXIS_MM2S* bzw. *AXI4_Stream-Master* (Master) Anschluss. Da es sich beim AXI-Stream um eine gerichtete Verbindung handelt, ist es entscheidend, dass immer ein Master-Port mit einem Slave-Port verbunden wird. Diese Verbindungen müssen manuell vorgenommen werden, da Vivado diese Zuordnungen nicht automatisch erkennen kann. Nachdem die Datenpfade korrekt verbunden wurden, schlägt Vivado in der Regel automatisch eine Verbindung für die zugehörigen Taktsignale und Reset-Signale vor. Wichtig ist dabei, das richtige Taktsignal auszuwählen, in diesem Fall 50 MHz, da die Filter in dieser Taktfrequenz ausgelegt sind.

32.3.5. Vollendetes Design

Sind alle Verbindungen korrekt vorgenommen, ergibt sich folgendes Blockdesign:

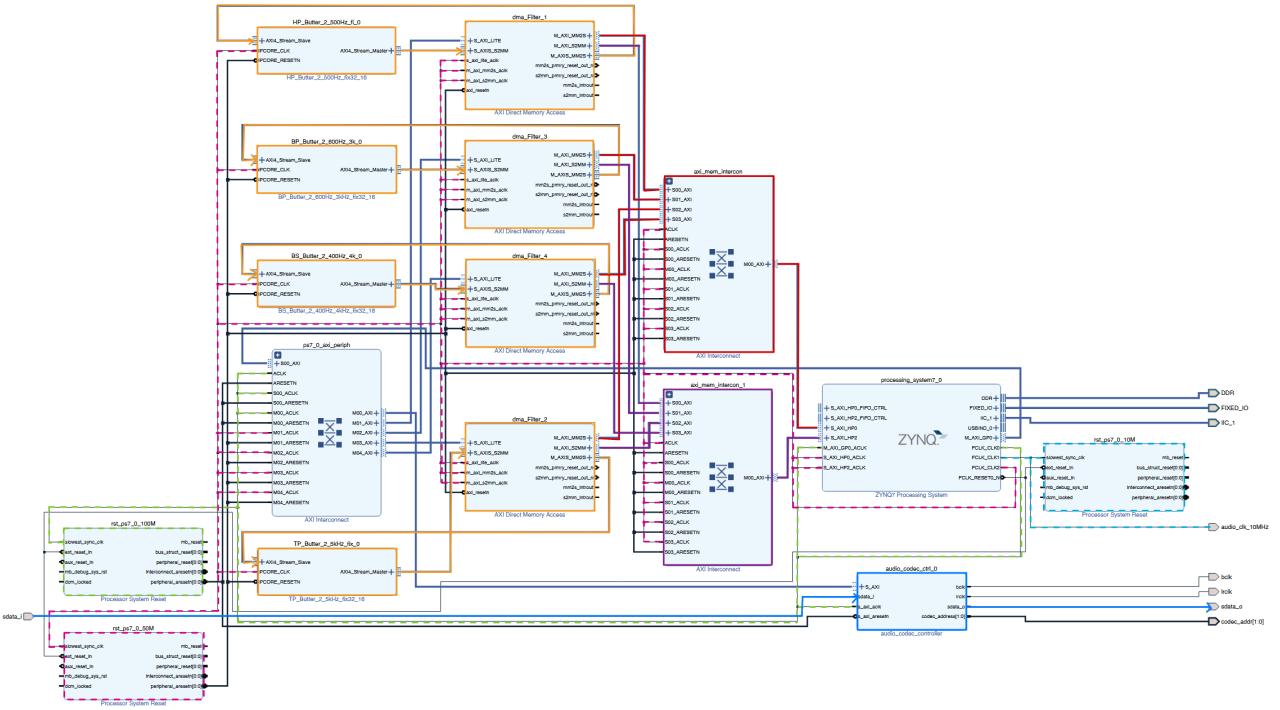


Abbildung 32.13.: Blockdesign

Die Clock-Signale sowie die wichtigsten Datenpfade wurden zusätzlich farblich hervorgehoben, um die Struktur übersichtlicher darzustellen. Nach erfolgreicher Validierung des Designs kann im nächsten Schritt der Bitstream generiert werden.

Am Ende besteht ein vollständiges Overlay aus zwei Dateien. Der generierten Bitstream-Datei (.bit) und der zugehörigen Hardware-Handoff-Datei (.hwh), welche die notwendigen Metadaten über das Design beinhaltet. Damit das Overlay von PYNQ korrekt erkannt und geladen werden kann, müssen beide Dateien denselben Basisnamen tragen mit Ausnahme der Dateiendung.

33. Funktionsweise des generierten Filter-Codes

Im Folgenden wird der generierte VHDL-Code der Filter näher erläutert. Die Beschreibung bezieht sich exemplarisch auf den Hochpassfilter. Da jedoch alle Filter nach dem gleichen Verfahren erzeugt wurden, lassen sich die Ausführungen sinngemäß auf sämtliche Filter übertragen. Die Modulnamen wurden automatisch vom HDL Coder generiert und unverändert übernommen. Der resultierende VHDL-Code zur Beschreibung der Filter-IP-Core ist aufgrund der zahlreichen verschachtelten Strukturen nur schwer überschaubar. Daher wird an dieser Stelle auf konkrete Codeausschnitte verzichtet. Stattdessen erfolgt eine allgemeine Darstellung des strukturellen Aufbaus und der funktionalen Gliederung der generierten Dateien. Hinzu kommt, dass der Code bei der Generierung verschiedene Optimierungsverfahren, insbesondere Pipelining, durchlaufen hat. Dadurch ist die ursprüngliche Filterstruktur im finalen VHDL-Code kaum noch erkennbar.

33.1. Struktur des generierten Codes

Das Top-Level-Modul bildet die oberste Instanz und integriert alle Funktionseinheiten. Die AXI4-Stream-Schnittstellen werden von zwei Modulen umgesetzt, einem AXI4-Stream Slave und einem Master. Der Slave empfängt die Eingangsdaten vom übergeordneten System und speichert sie zunächst in einem Eingangs-FIFO. Die eigentliche Signalverarbeitung findet im Design Under Test (DUT) statt. Hier werden die zwischengespeicherten Eingangsdaten durch die Biquad-Filtersektionen geleitet. Jede Sektion wird dabei durch ein separates Modul realisiert. Das DUT enthält ein Modul, das die gesamte Biquad-Filterstruktur beschreibt, die wiederum aus mehreren Sektionen bestehen kann. Die Daten werden Sektion für Sektion gefiltert, bis die gewünschte Filtercharakteristik erreicht ist.

Nach der Filterung werden die Ausgangsdaten in einem Ausgangs-FIFO zwischengespeichert, damit sie gepuffert zur Verfügung stehen. Ein spezieller FIFO sorgt für die korrekte Handhabung des TLAST-Signals, das im AXI4-Stream-Protokoll das Ende eines Frames oder Datenpakets signalisiert. Der AXI4-Stream Master gibt die gefilterten Daten schließlich zurück an das übergeordnete System. Ein zusätzliches Modul zur Reset-Synchronisation stellt sicher, dass ein globales Reset-Signal in allen Takt- und Logikbereichen zuverlässig übernommen wird.

Für den Datenverlauf bedeutet das, dass die Eingangsdaten über einen AXI4-Stream empfangen, gepuffert, durch den Biquad-Filter verarbeitet, erneut gepuffert und schließlich über AXI4-Stream ausgegeben werden. Der Datenfluss wird über Handshake-Signale gesteuert, um Datenverlust oder Überläufe zu vermeiden.

33.2. Umsetzung des Filters

33.2.1. Filter DUT

Wie bereits erwähnt, folgt der Code einer klaren Hierarchie. Das DUT übernimmt innerhalb dieses Designs mehrere Aufgaben. Es steuert, wann neue Daten aus dem Eingangs-FIFO gelesen werden und führt diese der Filter-Komponente zu. Nach der Filterung nimmt es die Ergebnisse entgegen, speichert sie im Ausgangs-FIFO und kümmert sich um alle notwendigen Steuersignale wie *Valid-/Ready*-Signale und Handshake-Mechanismen. Zudem stellt es sicher, dass die Filter-Komponente korrekt in die übergeordnete Takt- und Reset-Logik eingebunden ist.

33.2.2. Filter-Wrapper

Innerhalb des DUT wird die Komponente **src_Biquad_Filter** instanziiert. Diese Komponente fungiert als Wrapper und kapselt die eigentliche Filterkette. Sie nimmt die Eingangssignale entgegen, leitet sie unverändert weiter und gibt die Ergebnisse der Filterkette zurück an das DUT. Zudem stellt der Wrapper sicher, dass die Port-Zuordnung und Taktanbindung korrekt sind und das *clk_enable-Signal* richtig durchgereicht wird. Auf diese Weise wird die gesamte Filterlogik mit den Biquad-Sektionen innerhalb des Wrappers gebündelt. Es ist anzumerken, dass in der tatsächlichen Implementierung kein *clk_enable-Signal* aktiv genutzt wird, dennoch wurde es automatisch mitgeneriert.

33.2.3. Die Filterlogik

Wie bereits erwähnt, wird die Filterlogik exemplarisch am Hochpass-Filter erläutert. Es kann dabei zu minimalen Abweichungen bei anderen Filtern kommen, die grundsätzliche Funktionsweise ist jedoch bei allen Filtern vergleichbar.

Innerhalb des Wrappers wird eine gleichnamige Komponente instanziiert, die hierarchisch unterhalb des Wrappers angesiedelt ist. Hier befindet sich die eigentliche Filterlogik. Die Eingangsdaten (*dataIn)* werden zunächst in ein vorzeichenbehaftetes Signal (*dataIn_signed*) umgewandelt und im Register *sec0reg* zwischengespeichert. In der ersten Shifting- und Multiplikationsstufe (*sec0*) wird der Eingangswert durch Konkatenation mit Nullen nach links verschoben, um die gewünschte Fixpunkt-Skalierung zu erreichen. Das Ergebnis wird im Register *sec0mulreg* abgelegt, der relevante Bitbereich extrahiert (*sec0dtc*) und in *sec0out* zwischengespeichert. Von dort wird das Signal an die erste Biquad-Sektion weitergegeben.

Parallel dazu wird das Valid-Signal über eine Signalkette geführt, sodass die Gültigkeit der Daten in jeder Stufe erhalten bleibt. Die instanzierte Biquad-Sektion verarbeitet *sec0out* und liefert die gefilterten Daten (*sec1out*) sowie das zugehörige Valid-Signal (*sec1validout*). Anschließend folgt eine zweite Shifting- und Multiplikationsstufe (*sec1*), in der *sec1out* erneut skaliert und vorbereitet wird. Auch hier wird das Valid-Signal entsprechend weitergereicht. Für einen Filter zweiter Ordnung wird diese Sektion nur einmal aufgerufen, bei höheren Ordnungen werden mehrere Sektionen kaskadiert. Die Ausgabe wird konditional gesteuert. Wenn das Valid-Signal gültig ist, wird das gefilterte Ergebnis

an *dataOut* ausgegeben. Ein synchronisiertes Valid-Signal (*validOut*) zeigt an, wann die Daten gültig sind.

33.2.4. Filter-Sektion

Die Filter-Sektion bildet die eigentliche Biquad-Filterstruktur und enthält die entsprechenden Koeffizienten. Die Biquad-Sektion ist in transponierter Direct Form II umgesetzt. Sie erhält ein Eingangssample (*dataIn*) mit zugehörigem Gültigkeitssignal (*validIn*) und liefert das gefilterte Sample samt gültigem Ausgangssignal zurück.

Zunächst wird das Eingangssignal gepuffert. Anschließend wird es in drei Numerator-Zweigen parallel mit festen Koeffizienten multipliziert. Jeder Pfad besteht aus einem Vor-Register, dem Multiplizierer und einem Nach-Register. Die Denominator-Seite arbeitet mit zwei internen Zuständen (*state1* und *state2*), die über Multiplikationen und Addierer-Ketten rekursiv zurückgeführt werden. Diese Rückkopplung bildet den IIR-Charakter der Filterung.

Die kombinierte Summe wird auf den gewünschten Fixpunktbereich begrenzt und ausgegeben. Ein mehrstufiges Delay-Register sorgt dafür, dass das Ausgangssignal und das Valid-Signal zeitlich synchron anliegen. Zusammengefasst wird das Eingangssignal gepuffert, in drei parallelen Numerator-Zweigen mit festen Koeffizienten verarbeitet und mit den Rückkopplungswerten kombiniert. Die resultierende Summe wird skaliert und als gefiltertes Signal ausgegeben, während das Valid-Signal die zeitliche Korrektheit sicherstellt.

34. Echtzeitfilterung

Ursprünglich wurden die Filter als AXI4-Stream-fähige IP-Cores umgesetzt, um sie universell innerhalb von Vivado einsetzen zu können. Die bisherige Anwendung erfolgte über die Anbindung an einen AXI-DMA, um digitale Audioquellen zu filtern. Geplant war jedoch, denselben Filter (und gegebenenfalls weitere) auch für die Echtzeitverarbeitung analoger Audioquellen zu verwenden. Hierzu sollten die in Vivado verfügbaren LogiCORE-I^2S-Transmitter und -Receiver IPs in Kombination mit dem Audio-Codec eingesetzt werden.

In der vorgesehenen Konfiguration arbeitet der Audio-Codec-Controller als Master, d. h. er gibt die Taktsignale *bclk* und *lrclk* für die I^2S-Kommunikation vor. Der I^2S-Transmitter und -Receiver müssten in diesem Fall als Slaves fungieren. Genau hier trat das erste Problem auf. Laut offizieller Dokumentation unterstützen die I^2S IP-Cores keinen Slave-Modus, sondern sind ausschließlich für den Master-Betrieb vorgesehen. Da der Audio-Codec-Controller fest als Master konfiguriert ist ergab sich ein Konflikt. Mehere Master in einer I^2S-Verbindung würden unweigerlich zu Timing-Problemen führen.

Trotz dieser Einschränkung gelang es, durch gezielte Block-Property-Einstellungen in Vivado sowohl den I^2S-Receiver als auch den Transmitter im Slave-Modus zu betreiben, entgegen der offiziellen Angaben. Der Audio-Codec-Controller blieb dabei weiterhin Taktgeber. Die Initialisierung der I^2S-IP-Cores erfolgte über ein Jupyter Notebook, in dem die Register gemäß Datenblatt beschrieben wurden. Während für den Receiver die Samplerate und das Enable-Register gesetzt werden mussten, reichte beim Transmitter das Aktivieren über das Enable-Register aus.

34.1. Funktionstest & Filtereinbindung

In einem ersten Test wurde der Receiver direkt mit dem Transmitter über AXI-Stream verbunden. Diese Konfiguration funktionierte einwandfrei. Das Audiosignal wurde korrekt wiedergegeben, unabhängig davon, ob Receiver und Transmitter vor oder nach dem Codec-Controller eingebunden waren. Anschließend wurde der eigene Biquad-Filter-IP-Core zwischen Receiver und Transmitter geschaltet. Zwar blieb das Signal grundsätzlich vorhanden, war jedoch sehr leise und klang unverändert, eine effektive Filterung konnte nicht festgestellt werden.

Zur Fehlereingrenzung wurde eine vereinfachte Filter-IP-Core entwickelt, die nur als Bypass ohne jegliche Signalverarbeitung fungierte. Doch auch in diesem Fall blieb das Ausgangssignal stark gedämpft. Die Ursache lag nicht in der Filterlogik selbst.

Weitere Tests bestätigten, dass der vom HDL Coder erzeugte AXI-Datenpfad, bestehend aus TDATA, TVALID und TREADY, korrekt generiert wurden. Verschiedene Datenformate (signed/unsigned),

Skalierungsfaktoren, Gain-Blöcke und Bit-Shifts hatten keinen nennenswerten Einfluss auf die Lautstärke. Im Gegensatz dazu führte der Einsatz eines AXI4-Stream Data FIFO zwischen Receiver und Transmitter zu einer fehlerfreien, normal lauten Wiedergabe. Dies ließ den Verdacht aufkommen, dass der FIFO intern ein AXI-konformes Handshake-Verhalten sicherstellt, das von der eigenen IP nicht vollständig erfüllt wird.

Eine selbst erstellte IP mit identischer Buffertiefe (1024 Bit) brachte allerdings keine Verbesserung. Auch das optionale AXI-Signal TID, das bei den I^2S-IP-Cores zur Kanalunterscheidung dient, konnte in Simulink weder direkt erzeugt noch verarbeitet werden. Selbst ein manuelles Durchreichen außerhalb der IP zeigte keine Wirkung. Ein zusätzlicher FIFO hinter der IP, als Versuch, den Handshake zu stabilisieren, blieb ebenso erfolglos. Auch die Kombination von FIFO vor und nach der IP brachte keine Verbesserung, wodurch der Handshake als Fehlerursache weitgehend ausgeschlossen werden konnte. Der HDL Workflow Advisor bestätigte, dass alle AXI-Signale korrekt zugewiesen wurden und sogar eine explizite Steuerlogik für TREADY generiert wurde.

34.2. Ursache und Bewertung

Eine eindeutige Ursache für das fehlerhafte Verhalten konnte nicht ermittelt werden. Die Vermutung liegt jedoch nahe, dass die minimale AXI-Stream Konfiguration der vom HDL Coder generierten IP nicht alle Protokollanforderungen erfüllt, die der I^2S-Transmitter erwartet. Besonders die fehlende Unterstützung für optionale Signale wie TID und eine nicht vollständige Flusskontrolle könnten das Verhalten erklären. Um dieses Problem zu beheben, wäre ein ausgereifteres und weiter ausgeführtes Design erforderlich, das gegebenenfalls zusätzliche Signale berücksichtigt. Ein solches Design müsste unter Umständen manuell in VHDL oder Verilog implementiert werden und würde tiefgehende Kenntnisse des AXI-Protokolls, detaillierte Timinganalysen sowie ergänzende Simulationsschritte außerhalb von MATLAB/Simulink erfordern. Aufgrund der begrenzten Projektzeit und dem Fokus auf die eigentliche Filterimplementierung wurde entschieden, die Echtzeit-Anbindung über I^2S Receiver/Transmitter nicht weiterzuverfolgen. Stattdessen wird der Filtereinsatz über AXI-DMA demonstriert, die bereits erfolgreich mit digitalen Quellen funktioniert.

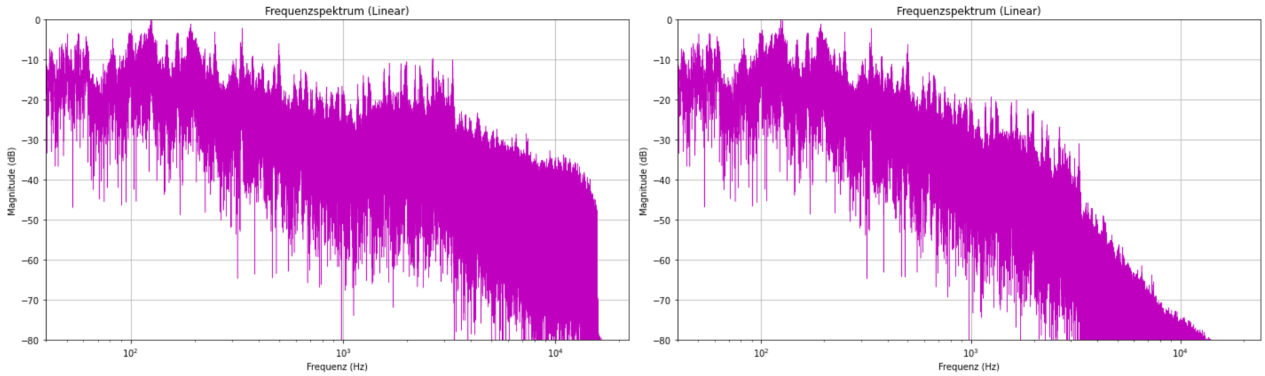
34.3. Analoge Audioquellen

Da die geplante Echtzeitfilterung analoger Signale nicht zeitnah umsetzbar war, wurde das Design zur Filterung digitaler Quellen entsprechend erweitert. Mithilfe des auf dem PYNQ-Z2-Board integrierten Audio-Codecs ist es nun möglich, analoge Audiosignale aufzunehmen und als Wav-Datei auf dem Board zu speichern. Diese digitale Datei kann anschließend vom bestehenden Design verarbeitet und gefiltert werden. Dadurch ist es trotz der Einschränkungen bei der Echtzeitverarbeitung möglich, analoge Audioquellen zu filtern, jedoch mit einer zwischengelagerten, nicht-echtzeitfähigen Verarbeitungskette.

35. Aufbau und Funktion der Lehrdemonstration

Ein zentrales Element der Lehrdemonstration ist ein vorgefertigtes Jupyter-Notebook, mit dem sich die verschiedenen Filter einfach bedienen lassen. Innerhalb dieses Notebooks können analoge Audiosignale über die Eingänge des Boards aufgenommen und sowohl im Notebook selbst als auch über den analogen Audioausgang des PYNQ-Z2 wiedergegeben werden. Darüber hinaus besteht die Möglichkeit, sowohl die aufgenommenen Audiodaten als auch externe Audiodateien mit einem der vier implementierten Filter zu verarbeiten. Nach der Auswahl eines Filters wird das gefilterte Signal gespeichert und kann angehört oder weiter analysiert werden.

Da sowohl das Audiosignal vor als auch nach der Filterung gespeichert wird, besteht die Möglichkeit, beide Versionen direkt miteinander zu vergleichen. Neben dem reinen Abhören der Audiodateien kann zusätzlich ein definierter Ausschnitt des Frequenzspektrums vor und nach der Filterung visualisiert werden. Dies ermöglicht eine anschauliche Analyse der Signalveränderung im Frequenzbereich und macht die Wirkung des gewählten Filters unmittelbar sichtbar. Auf diese Weise wird die digitale Signalverarbeitung nicht nur hörbar, sondern auch grafisch nachvollziehbar.



(a) Frequenzspektrum vor dem Filtern (b) Frequenzspektrum nach der Tiefpassfilterung

Abbildung 35.1.: Frequenzspektrum vor und nach dem Filtern

In den Abbildungen ist nicht das vollständige Frequenzspektrum dargestellt. Stattdessen wird lediglich ein Ausschnitt von den ersten 30 Sekunden des linken Audiokanals gezeigt. Dies wurde bewusst so gewählt, da eine vollständige Analyse des gesamten Signals rechnerisch zu aufwendig wäre, um sie im Jupyter Notebook durchzuführen. Die Darstellung erfolgt in normierter Form. Die Amplitudenwerte werden so skaliert, dass der jeweils höchste Wert bei 0 dB liegt. Dadurch kann es zwar zu Unterschieden in der absoluten Lautstärke zwischen verschiedenen Plots kommen, jedoch bleibt der relative Vergleich zwischen verschiedenen Frequenzverläufen jederzeit möglich.

35.1. Pynq Overlays

Zur Nutzung individuell entwickelter Hardwaredesigns auf dem PYNQ-Z2-Board werden Overlays verwendet. Diese bestehen aus einer Bitstream-Datei (.bit) und der zugehörigen Hardware Handoff-Datei (.hwh), die zuvor mit Vivado generiert werden. In einigen PYNQ-Dokumentationen wird teilweise auch auf .tcl-Dateien verwiesen. Im Rahmen der hier eingesetzten PYNQ-Version hat sich gezeigt, dass diese Datei nicht erforderlich ist. Das Overlay lässt sich vollständig mit der .bit- und der .hwh-Datei verwenden.

Die eigentliche Ansteuerung der Hardware erfolgt über das auf dem PYNQ-Board vorinstallierte Python-Modul pynq. Dieses Modul stellt eine spezielle Bibliothek bereit, mit der sich aus Python heraus direkt auf die programmierbare Logik (PL) zugreifen lässt. Dadurch können AXI-DMA-Schnittstellen genutzt, Register beschrieben oder externe IP-Cores über vordefinierte Treiber angesprochen werden. Die Steuerung der Hardware erfolgt in Python über Jupyter Notebooks, die direkt auf dem PYNQ-Board ausgeführt werden. Der Zugriff auf die Notebooks erfolgt remote über Geräte im selben Netzwerk.

Zum einen wird das Modul pynq.lib.dma verwendet, das den Zugriff auf die AXI-DMA-Schnittstellen der programmierbaren Logik ermöglicht. Über dieses Modul können mit wenigen Python-Befehlen Daten an die Filter-IP-Cores gesendet und die verarbeiteten Daten wieder empfangen werden. Die Übertragung wird dabei automatisch vom Modul verwaltet, sodass keine manuelle Kontrolle über Adressen notwendig ist

Zum anderen kommt das Modul pynq.lib.audio zum Einsatz. Dieses Modul enthält den Treiber für den auf dem PYNQ-Z2-Board verbauten ADAU1761-Audiocodec und ermöglicht es, die analogen Audioein- und -ausgänge des Boards direkt aus Python heraus anzusteuern. Über den Treiber können zentrale Funktionen wie das Setzen der Abtastrate, das Aktivieren oder Deaktivieren von Ein- und Ausgängen gesteuert werden. Darüber hinaus bietet das Modul die Möglichkeit, Audiosignale über die analogen Eingänge aufzunehmen.

Beim Versuch, die Abtastrate von 48 kHz auf 44,1 kHz umzustellen, zeigte sich jedoch, dass die Aufnahmen trotz korrekter Konfiguration weiterhin mit 48 kHz durchgeführt wurden. Zwar wurde in der erzeugten WAV-Datei die Samplingrate ordnungsgemäß mit 44,1 kHz eingetragen, doch beim Abhören war ein deutlich tieferer Pitch zu erkennen, was ein Anzeichen dafür ist, dass die tatsächliche Aufnahmefrequenz höher lag als angegeben. Bei der direkten Wiedergabe über die Hardware blieb dieser Fehler zunächst unbemerkt. Sobald die Aufnahme über das Notebook abgespielt wurde, war der Pitch sofort hörbar. Auch externe WAV-Dateien mit 44,1 kHz klangen über den Hardware-Ausgang zu schnell, wenn dieser auf 44,1 kHz eingestellt wurde, was ebenfalls auf eine Diskrepanz zwischen eingestellter und tatsächlich verwendeter Abtastrate hinweist. Aus diesem Grund wurde das gesamte Filterdesign konsequent auf eine Samplingrate von 48 kHz ausgelegt. Um dennoch Audiodateien mit niedrigeren Abtastraten verarbeiten zu können, werden diese vor der Weiterverarbeitung entsprechend hochgesampelt.

35.2. Funktionsweise des Notebooks

Zur besseren Übersicht und vereinfachten Bedienung wurden alle wesentlichen Prozesse, wie Datenübertragung, Speichern und Visualisierung, in einer separaten Python-Datei als einheitliche Funktionssammlung zusammengeführt. Diese Struktur ermöglicht es, komplexe Abläufe mit nur wenigen, klaren Funktionsaufrufen direkt im Jupyter Notebook auszuführen.

Die wichtigste Rolle spielt dabei die Übertragung der Audiodaten an den DMA, der diese durch den Filter-IP-Core leitet und anschließend wieder zurückgibt. Die grundlegende Datenübertragung an den DMA erfolgt dabei relativ einfach und orientiert sich im Prinzip an dem Ablauf, wie er auch in den offiziellen PYNQ-Tutorials und der PYNQ-Dokumentation beschrieben ist.

35.3. Initialisierung des DMA-Kanals

Am Anfang werden alle notwendigen Bibliotheken importiert, einschließlich der PYNQ-spezifischen Module wie *Overlay*, *allocate* sowie die Treiberbibliothek *pynq.lib.audio*. Anschließend wird das entsprechende DMA-IP-Core aus dem zuvor geladenen Overlay angesprochen und in einer Variable gespeichert. Die DMA-Instanz verfügt über zwei Kanäle. Für jeden der vier Filter wird eine eigene DMA-Instanz erzeugt und jeweils den zugehörigen Variablen zugewiesen.

```
from pynq import Overlay
from pynq import allocate
from pynq.lib.audio import AudioADAU1761

ol = Overlay("Audio_quad_Filter_v7.bit")

dma_Filter_1 = ol.dma_Filter_1
dma_Filter_1_send = ol.dma_Filter_1.sendchannel
dma_Filter_1_recv = ol.dma_Filter_1.recvchannel
```

35.4. DMA Übertragung

Eine Zentrale Funktion ist die *Transmission(input_data,ip_buffer,dma,dma_send, dma_recv)*. Diese Funktion wird für jede übertragung aufgerufen und übernimmt die Kernaufgaben für eine vollständige übertragung.

Bevor die Audiodaten an die DMA-Schnittstelle übergeben werden können, müssen sie zunächst normiert und in ein kompatibles Datenformat umgewandelt werden. Dies ist notwendig, da die AXI-DMA-Schnittstelle nur mit bestimmten Datentypen arbeiten kann.

```
# Festlegen der Größen
buffer_size = int(ip_buffer)
```

```
input_data = FormatChange(input_data)
data_size = int(len(input_data))
```

Die DMA-Instanz verfügt über zwei Kanäle, den *sendchannel* zum Senden von Daten und den *recvchannel* zum Empfangen der verarbeiteten Daten. Diese beiden Kanäle werden für die gewählte DMA-Instanz initialisiert. Bevor die Audiodaten in den input_buffer geladen werden, wird dieser zunächst vollständig mit Nullwerten gepuffert. Der Hintergrund ist, dass beim Anlegen des Buffers mittels *allocate()* dieser zunächst undefinierte Werte enthält. Falls die zu übertragende Datenmenge kleiner als die definierte Buffergröße ist, bleiben einzelne Felder leer. Der AXI-DMA prüft jedoch bei jedem Transfer, ob gültige Daten im Puffer vorhanden sind. Sind nicht alle Speicherbereiche korrekt befüllt, kann der DMA in einen Fehlerzustand übergehen. In diesem Fall wird der gesamte Datenpfad blockiert, und es sind keine weiteren Übertragungen möglich, bis ein manueller Reset durchgeführt wird. Um diesem vorzubeugen, wird der *input_buffer* vor dem eigentlichen Datentransfer vollständig mit Nullen überschrieben. Anschließend werden die eigentlichen Nutzdaten an den Anfang des Buffers geschrieben. Nach der Übertragung wird der Puffer anhand der bekannten Datenlänge wieder korrekt zurückgeschnitten.

```
# Padding
pad = np.zeros(ip_buffer)
pad_frame = FormatChange(pad)

# Leere Buffer
input_buffer = allocate(shape=(buffer_size,), dtype=np.uint32)
output_buffer = allocate(shape=(buffer_size,), dtype=np.uint32)

# Padding Inputbuffer
input_buffer[:] = pad_frame

# Laden der Daten in Inputbuffer
input_buffer[: data_size] = input_data
```

Die Datenübertragung wird mithilfe der Methode *transfer()* sowohl für den Sende- als auch für den Empfangskanal gestartet. Dadurch wird der Übertragungsvorgang an den DMA angestoßen. Nach dem Start der Übertragung wird mit *wait()* auf den Abschluss des Transfers gewartet. Während dieser Phase ist die DMA-Schnittstelle blockiert,

```
# Senden un Empfangen der Daten
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()
```

Nach dem Empfang der Daten über den DMA werden diese wieder in das ursprüngliche Format zurückkonvertiert. Falls der Eingabepuffer aufgrund von Padding mit Nullen aufgefüllt wurde, werden diese nach dem Empfang anhand der bekannten Originallänge entfernt, sodass nur die tatsächlich relevanten Audiodaten weiterverarbeitet werden.

```
# Umrechnen der Empfangenen Daten
output_data = np.array(output_buffer[: data_size]).view(np.int32)
```

35.5. Öffnen und Einlesen von Wav-Dateien

Zur Verarbeitung von Audiodaten werden zunächst Wav-Dateien geöffnet und eingelesen. Dies geschieht über die Funktion *read_wav()*, die mithilfe des Python-Moduls wave Zugriff auf die Audiodaten ermöglicht. Die Funktion basiert auf einem Beispiel aus den PYNQ-Audio-Demonstrationen, die auf dem Board vorinstalliert sind, und wurde entsprechend angepasst. Beim Einlesen der Datei werden zunächst grundlegende Metainformationen wie Anzahl der Kanäle, Abtastrate, Anzahl der Frames sowie die Sample-Breite ausgelesen. Anschließend werden die Rohdaten byteweise in ein Array geladen und abhängig von der Kanalanzahl und Samplegröße korrekt umstrukturiert. Am Ende gibt die Funktion sowohl die Audiodaten als auch die zugehörigen Metadaten (Frame-Anzahl, Kanalzahl, Samplingrate, Sample-Breite) zurück.

```
def read_wav(wav_path):
    with wave.open(wav_path, 'r') as wav_file:
        raw_frames = wav_file.readframes(-1)
        num_frames = wav_file.getnframes()
        num_channels = wav_file.getnchannels()
        sample_rate = wav_file.getframerate()
        sample_width = wav_file.getsampwidth()

    temp_buffer = np.empty((num_frames, num_channels, 4), dtype=np.uint8)
    raw_bytes = np.frombuffer(raw_frames, dtype=np.uint8)
    temp_buffer[:, :, :sample_width] = raw_bytes.reshape(-1, num_channels,
                                                         sample_width)
    temp_buffer[:, :, sample_width:] = \
    (temp_buffer[:, :, sample_width-1:sample_width] >> 7) * 255
    frames = temp_buffer.view('<i4').reshape(temp_buffer.shape[:-1])

    print("Frames:",len(frames), "Channels:", num_channels, "Sample Rate:",sample_rate, "Sample W
    return frames, num_frames, num_channels, sample_rate, sample_width
```

35.6. Schreiben und Speichern von Wav-Dateien

Nach der Verarbeitung der Audiodaten können die Ergebnisse als Datei gespeichert werden. Dies geschieht über die Funktion *save_to_24bit_wav()*, die die beiden Audiokanäle entgegennimmt und diese gemeinsam in eine Stereo-WAV-Datei schreibt. Die Funktion ist in ihrer Struktur an *read_wav()* angelehnt und übernimmt das Schreiben der Daten im passenden Format. Das Schreiben erfolgt über das Python-Modul *wave*, wobei die Daten als flache Bytefolge übergeben werden. Die resultierende

Datei kann anschließend direkt im Notebook abgespielt oder zur externen Weiterverarbeitung verwendet werden.

```
def save_to_24bit_wav(chan_l, chan_r, sample_rate, path):
    frames = np.stack((chan_l, chan_r), axis=1)
    max_val = 2**23  # 24-bit max signed int
    frames = np.clip(frames, -1.0, 1.0)
    frames_int = (frames * max_val).astype(np.int32)

    # In Bytes umwandeln
    temp_bytes = frames_int.reshape((*frames.shape, 1)).view(np.uint8)
    raw_bytes = temp_bytes[:, :, :3].reshape(-1)

    with wave.open(path, 'wb') as wav_out:
        wav_out.setnchannels(frames.shape[1])
        wav_out.setsampwidth(3)  # 24-bit
        wav_out.setframerate(sample_rate)
        wav_out.writeframes(raw_bytes.tobytes())
```

35.7. Paketaufteilung

Wie bereits bekannt, ist die Puffergröße der Filter-IP-Cores auf 2^{20} Bit festgelegt. Nach dem Einlesen einer .wav-Datei kann es jedoch vorkommen, dass die Audiodaten (Frames) länger sind als der verfügbare Puffer. Um dennoch eine korrekte Filterung beliebiger Signallängen zu ermöglichen, werden die Daten vor der Übertragung in kleinere Teilstücke zerlegt. Dies erfolgt über die Funktion *Split2Packets()*, die das Array mit den Audiodaten in mehrere Pakete unterteilt, jeweils mit einer maximalen Größe, die dem festgelegten Puffer entspricht.

```
def Split2Packets(data,packet_size):
    packets = []
    for i in range(0, len(data), packet_size):
        packet = data[i:i+packet_size]
        packets.append(packet)
    return packets
```

Die Funktion *send2receive()* nutzt zunächst *Split2Packets()*, um die Eingangsdaten in kleinere Teilstücke aufzuteilen. Für jedes dieser Datenpakete wird anschließend die Übertragung mithilfe der Funktion *Transmission()* durchgeführt. Die dabei empfangenen, gefilterten Daten werden jeweils aneinandergehängt und in einem Ausgabearray gesammelt. Auf diese Weise entsteht am Ende ein vollständiges gefiltertes Signal, das dieselbe Länge wie das ursprüngliche Eingangsframe besitzt.

```
def send2receive(Data_In, dma, dma_send, dma_recv):
    ip_buffer = 2**20
    # Filtern Kanal
```

```
    Data_Out = []

    # Zerteilung und Übertragung in Packeten
    Packets = Split2Packets(Data_In, ip_buffer)
    anz_trans = 0
    for packet in Packets:
        result = Transmission(packet, ip_buffer,dma, dma_send, dma_recv)
        Data_Out.extend(result)
        anz_trans = anz_trans + 1
    print(anz_trans, "Transmissions")
    return Data_Out
```

35.8. vollständige Filter-Anwendung

Die Funktion *UseFilter()* bildet den zentralen Ablauf zur Anwendung eines digitalen Filters auf eine Audiodatei. Sie automatisiert dabei alle notwendigen Schritte, vom Einlesen der Datei bis zur Speicherung des gefilterten Ergebnisses.

Zu Beginn wird die gewählte Filterinstanz ausgewählt. Anschließend wird die angegebene .wav-Datei über die Funktion *read_wav()* geöffnet. Dabei werden sowohl die Audiodaten der beiden Kanäle als auch relevante Metainformationen ausgelesen. Anschließend erfolgt eine Normierung der Audiodaten um eine saubere Signalverarbeitung sicherzustellen.

Falls die Samplerate der Eingangsdaten nicht 48 kHz beträgt, was häufig bei Standard-Audiodateien mit 44,1 kHz der Fall ist, wird automatisch ein Upsampling auf 48 kHz durchgeführt. Dies ist erforderlich, da die Filter für diese Abtastrate ausgelegt wurden. Auf diese weise wird das richtige Verhalten des Filters sichergestellt.

Im nächsten Schritt wird das normierte Audiosignal für jeden Kanal getrennt an den zuvor ausgewählten Filter übergeben. Die Übertragung und Filterung wird über die Funktion *send2receive()* abgewickelt, die das Signal in Pakete unterteilt, überträgt, empfängt und anschließend wieder zusammensetzt.

Zuletzt werden die gefilterten Daten beider Kanäle mithilfe der Funktion *save_to_24bit_wav()* wieder zu einer Stereo-WAV-Datei zusammengeführt und unter dem angegebenen Ausgabedateinamen gespeichert.

```
def UseFilter(in_Name, out_Name,Filter):
    import time
    start = time.time()
    dma = Filter[0]
    dma_send = Filter[1]
    dma_recv = Filter[2]
    [frames, num_frames, channels, Fs, Fw] = read_wav(in_Name)
    # Read Data
```

```
data_l = Normierung(frames[:,0])
data_r = Normierung(frames[:,1])
if Fs != 48000:
    data_l = resample_poly(data_l,48000, Fs)
    data_r = resample_poly(data_r,48000, Fs)
    print("up-sampled: 44.1 -> 48kHz")
    Fs = 48000
print("Start Sending left Channel")
Data_Out_L = send2receive(data_l, dma, dma_send, dma_recv)
print("Start Sending right Channel")
Data_Out_R = send2receive(data_r, dma, dma_send, dma_recv)
print("Saving File")
save_to_24bit_wav(Data_Out_L, Data_Out_R, Fs, out_Name)
end = time.time()
print("Finished")
print(f"Dauer: {end - start:.2f} Sekunden")
```

35.9. Kaskadierung der Filter

Neben der Standardfunktion *UseFilter()* steht eine erweiterte Variante zur Verfügung, die es ermöglicht, dasselbe Audiosignal mehrfach durch denselben Filter zu leiten. *UseFilterCascade()* erlaubt es, über einen Parameter festzulegen, wie oft die Filterung wiederholt werden soll. Hintergrund ist, dass in jedem Filter-IP-Core lediglich ein einzelnes Biquad-Segment implementiert ist. Durch die wiederholte Anwendung desselben Filters lässt sich somit eine Kaskadierung mehrerer Biquads realisieren. Dies führt zu einer Erhöhung der effektiven Filterordnung und damit zu einer steileren Flankensteilheit im Frequenzgang, ein Effekt, der deutlich hör- und sichtbar wird.

Diese Funktion dient ausschließlich zu Demonstrationszwecken, bei der gezeigt werden soll, wie sich die Eigenschaften eines Filters durch mehrfache Anwendung bzw. Kaskadierung verändern lassen.

36. Fazit und Ausblick

Die Umsetzung digitaler Filter auf FPGA-Hardware ist komplex und erfordert in der Regel fundierte Kenntnisse in VHDL sowie ein tiefes Verständnis der Zielarchitektur. Der modellbasierte Workflow mit MATLAB/Simulink und dem *HDL Coder* reduziert diesen Aufwand, da aus einem Simulink-Modell automatisch synthesefähiger VHDL-Code generiert wird. Dadurch können typische Fehlerquellen frühzeitig erkannt und behoben werden. Besonders vorteilhaft ist die strukturierte Anleitung durch den *HDL Workflow Advisor*, der den Code schrittweise an die Zielplattform anpasst. Anforderungen wie AXI4-Stream-Kompatibilität oder Taktmanagement werden automatisch berücksichtigt. Auf dieser Grundlage konnten alle vier IIR-Filter als parametrisierte Simulink-Modelle entworfen und als IP-Cores generiert werden.

Die Integration in Vivado erfolgte über den *Block Design Editor*, dessen grafische Oberfläche eine intuitive Verbindung der Module ermöglicht. Besonders in der iterativen Entwicklungsphase erwies sich dieser modulare Aufbau als praktikabel. Ein geplanter Echtzeitbetrieb über die I²S-Schnittstelle des ADAU1761 scheiterte jedoch an Kompatibilitätsproblemen zwischen den automatisch generierten Filter-IP-Cores und den bestehenden I²S-Komponenten. Die genaue Ursache konnte aufgrund der abstrahierten Struktur der HDL-Coder-Ausgabe nicht eindeutig identifiziert werden.

Stattdessen wurde eine alternative Lösung zur Verarbeitung zuvor aufgezeichneter Audiodaten realisiert. Dabei wurde die Fähigkeit des PYNQ-Z2 genutzt, analoge Quellen über den integrierten Audio-Codec aufzunehmen. Das Design in Vivado konnte entsprechend angepasst werden, um diese Funktionalität zu unterstützen. Die Filterung erfolgt anschließend über ein eigens entwickeltes Jupyter-Notebook, das auf dem PYNQ-Z2 ausgeführt wird. Die Benutzeroberfläche ermöglicht eine unkomplizierte Anwendung der digitalen Filter auf gespeicherte Audiosignale, sowohl aus digitalen als auch aus analogen Quellen.

Trotz einzelner technischer Herausforderungen konnte somit ein funktionales Gesamtsystem realisiert werden, das sich hervorragend für Demonstrations- und Lehrzwecke eignet.

Für eine zukünftige Weiterentwicklung des Projekts bietet sich die Möglichkeit, an den etablierten Filterdesigns anzuknüpfen und diese funktional zu erweitern. Ein vielversprechender Ansatz wäre die Ergänzung eines Filters um eine AXI4-Lite-Schnittstelle, über die sich Register zur dynamischen Übergabe von Filterkoeffizienten ansteuern lassen. In Kombination mit einer funktionierenden Echtzeitfilterung könnte so ein anpassbarer Echtzeitfilter realisiert werden, dessen Charakteristik sich während des Betriebs flexibel verändern lässt. Dies würde das System nicht nur deutlich vielseitiger machen, sondern auch neue Einsatzmöglichkeiten in interaktiven oder adaptiven Audioszenarien erschließen.

37. Gegenüberstellung zweier Implementierungsmethoden digitaler IIR-Filter

Parallel zu dieser Bachelorarbeit wurde eine weitere Arbeit mit ähnlicher Fragestellung durchgeführt. Beide Arbeiten beschäftigen sich mit der praktischen Umsetzung digitaler IIR-Filter, unterscheiden sich jedoch grundlegend in der gewählten Implementierungsplattform. Während sich die eine Arbeit auf die softwarebasierte Realisierung mit Mikrocontrollern (ESP32, Arduino, LyraT) konzentriert, steht in dieser Arbeit die hardwarebasierte Umsetzung auf einem FPGA-System (PYNQ-Z2) im Mittelpunkt.

Die Mikrocontroller-Implementierung erfolgt direkt in C++ (Arduino) bzw. MicroPython. Die Biquad-Filter werden dabei auf Basis ihrer Differenzengleichungen manuell in Code umgesetzt. Verschiedene Strukturformen wie Direktform I, Direktform II und deren transponierte Variante (TDF2) wurden explizit implementiert. Besonderheiten wie die manuelle Verwaltung interner Zustände oder Verzögerungsglieder sowie Optimierungen (z. B. durch ´**slots**´ in MicroPython oder native Kompilierung einzelner Methoden) tragen zur Effizienzsteigerung bei.

Während C++ eine vergleichsweise performante Umsetzung ermöglicht und eine Echtzeitverarbeitung über die I^2S-Schnittstelle grundsätzlich erlaubt, ist MicroPython aufgrund seiner Interpreterstruktur deutlich langsamer. Zusätzlich verhinderten fehlende I^2S-Treiber in MicroPython eine stabile Audioverarbeitung in Echtzeit. Daher beschränkte sich die MicroPython-Implementierung auf die Filterung zuvor gespeicherter WAV-Dateien.

Im Gegensatz dazu basiert die FPGA-Implementierung auf einem modellbasierten Entwicklungsansatz mittels MATLAB/Simulink und dem HDL Coder. Die Filterstrukturen, insbesondere Direct Form II transponiert mit Pipelining, werden als Simulink-Modelle aufgebaut und anschließend in synthesefähigen VHDL-Code überführt. Die resultierenden IP-Cores werden über AXI4-Stream-Schnittstellen in ein Vivado-Projekt eingebunden. Die Verarbeitung erfolgt hardwareseitig, was eine hohe Parallelität und Echtzeitfähigkeit erlaubt. Die PYNQ-Plattform ermöglicht darüber hinaus eine komfortable Steuerung und Visualisierung über Jupyter Notebooks mit Python. Ein geplanter Echtzeitbetrieb über die I^2S-Schnittstelle des integrierten Audio-Codecs ADAU1761 konnte jedoch aufgrund von Kompatibilitätsproblemen zwischen den automatisch generierten AXI-Strukturen und den bestehenden I^2S-Komponenten nicht umgesetzt werden.

Insgesamt zeigen beide Arbeiten komplementäre Lösungswege auf. Die Mikrocontroller-Variante punktet durch Einfachheit, Offenheit und geringe Einstiegshürden, während die FPGA-basierte Lösung eine leistungsstarke und skalierbare Architektur für komplexe Anwendungen bereitstellt. Beide Umsetzungen verdeutlichen unterschiedliche Strategien zur Realisierung digitaler Filter und bieten eine fundierte Grundlage für zukünftige Lehr-, Forschungs- oder Entwicklungsprojekte.