Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Sener Sahin

Entwicklung einer Audiocodec-Platine und Audiosignalverarbeitung auf einem FPGA mit High-Level Synthese

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Sener Sahin

Entwicklung einer Audiocodec-Platine und Audiosignalverarbeitung auf einem FPGA mit High-Level Synthese

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

Betreuender Prüfer : Prof. Dr.Ing. Lutz Leutelt
Zweitgutachter : Prof. Dr. Ulrich Sauvagerd

Abgegeben am 22. Juni 2018

Sener Sahin

Thema der Bachelorthesis

Entwicklung einer Audiocodec-Platine und Audiosignalverarbeitung auf einem FPGA mit High-Level Synthese

Stichworte

Audiocodec, Codec, PCM3006, Signalverarbeitung, Platine, FPGA, VHDL, Xilinx Vivado, Xilinx Vivado HLS, High-Level Synthese

Kurzzusammenfassung

In dieser Arbeit wird eine Audiocodec-Platine für das Labor für Digitaltechnik der Hochschule für Angewandte Wissenschaften (HAW) in Hamburg entwickelt und ein VHDL Programm für die Ansteuerung der Platine mit einem FPGA geschrieben. Zudem werden erste Erfahrungen mit der Software Vivado HLS von Xilinx gesammelt.

Sener Sahin

Title of the paper

Development of an Audiocodec-Board and signal processing on an FPGA with High-Level Synthesis

Keywords

Audiocodec, Codec, PCM3006, Signal processing, PCB, FPGA, VHDL, Xilinx Vivado, Xilinx Vivado HLS, High-Level Synthesis

Abstract

This bachelor thesis describes the development of an audiocodec pcb for the labour of digital technology on Hamburg University of Applied Sciences (HAW). For taht a vhdl application is developed to control the audiocodec pcb. Then first experience with the Xilinx Software Vivado HLS are collected.

Inhaltsverzeichnis

Abkürzungsverzeichnis

FPGA Field Programmable Gate Array

VHDL Very High Speed Integrated Circuit Hardware Description Language

ADU Analog-Digital-Umsetzer

DAU Digital-Analog-Umsetzer

OpAmp Operationsverstärker

ModSys-Board Modularsystem

I/O Input Output

MSB Bitwertigkeit, Most Significant Bit

MSB Bitwertigkeit, Least Significant Bit

FF Flipflop

SYSCLK Systemtakt

CDCCLK Codectakt

BCKIN Bit-Takt

LRCIN Wort-Takt

FIR finite impulse response

HLS High-Level Synthese

I2C Inter-Integrated Circuit

I2S Inter-IC Sound

PWM Pulsweitenmodulation

PCM Puls-Code-Modulation

IC Integrated Circuit

SPU Serien-Parallel-Umsetzer

PSU Parallel-Serien-Umsetzer

SNR Rausch-Signal-Abstand

PCB Printed Circuit Board

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

Diese Bachelorthesis beschreibt die Entwicklung einer Audiocodec-Platine und Signalverarbeitung auf einem FPGA mit High-Level Synthese. In diesem ersten Kapitel wird die Motivation der Arbeit erläutert. Als dann wird kurz den Inhalt der nachfolgenden Kapitel erläutert.

1.1. Motivation

Im Zeitalter der Digitalisierung wird die Signalverarbeitung auf der digitalen Ebene immer wichtiger. Jedoch liegen viele Signale in analoger Form vor und die Sinne des Menschen nehmen die Umwelt analog wahr. Die Verbindung der analogen und digitalen Welt wird mit hochwertigen Analog-Digital und Digital-Analog-Umsetzer ermöglicht. Der Mensch verfügt über fünf Sinne; Riechen, Schmecken, Fühlen, Sehen und Hören. Im speziellen das Gehör nimmt die Umgebung durch Schall war. Um den analogen Schall zu digitalisieren wird ein Audiocodec verwendet. Field Programmable Gate Arrays, kurz FPGA, sind Hardware, die sich programmieren, bzw. sich Konfigurieren lassen und somit anders als die Hardware, bei der die Funktionen zum größten Teil festgestellt ist. Mit FPGAs können sich von simplen Funktion, bis zu komplexen Konstrukten konfigurieren lassen. Dabei verdrängt der FPGA zunehmend mehr den klassischen Prozessors[5][6][7].

Mit dieser Arbeit soll das Digitaltechnik Labor der Fakultät der Technik und Informatik mit einer Platine erweitert werden. Die Platine soll modular an das bestehende ModSys-Baseboard angepasst werden, sodass die zu entwickelnde Platine nahtlos an das ModSys-Baseboard angebracht werden kann.

Die zu entwickelnde Audiocodec-Platine soll analoge Audiosignale ins Digitale umgesetzt, digital Verarbeitet und zum Schluss ins analoge umgesetzt und ausgegeben werden. Die Platine soll im Digitaltechnik-Labor in Einsatz kommen. Die digitale Signalverarbeitung soll über einen FPGA erfolgen.

1.2. Kapitelübersicht

Nachdem dieses Kapitel die Motivation aufgezeigt hat, beschreibt das zweite Kapitel einige Grundlagen, die für die Bearbeitung dieser Arbeit für notwendig erachtet wurden. Die vorliegende Arbeit ist in sieben Kapitel unterteilt. Im Folgenden werden die Inhalte erläutert.

In Kapitel 1 wird die Motivation aufgezeigt. Weiterhin wird das Ziel dieser Arbeit beschrieben.

In Kapitel 2 werden erforderliche Grundlagen behandelt. Im Einzelnen wird einige Grundlagen zur Tontechnik beschrieben. Des Weiteren werden analoge Schaltungen zur Verstärkung von Signalen beschrieben. Weiterhin wird das im Labor verwendete ModSys-Baseboard 2 erläutert. Anschließend werden digitale Grundschaltungen in VHDL behandelt. Zum Schluss werden zwei Audioeffekte vorgestellt.

In Kapitel 3 werden die Anforderungen an die Arbeit analysiert. Dabei werden die Anforderungen in vier Unterkapitel geteilt. Zuerst werden die Anforderungen an den Audiocodec, dann an die Platine, anschließend an das VHDL-Modell und zum Schluss an den Audioeffekt gestellt.

In Kapitel 4 werden einzelne Konzepte entwickelt. Als erstes wird das Gesamtsystem betrachtet. Anschließend wird ein der Audiocodec ausgewählt und einige Eigenschaften aufgezählt. In Anschluss werden die Konzepte für die Platine entwickelt. Anschließend wird die Konzepte für das VHDL-Grundstruktur behandelt. Zum Schluss wird das Konzept für die Entwicklung von den Audioeffekten beschrieben.

In Kapitel 5 wird die analoge Schaltung entworfen und eine passende Platine entwickelt. Im Anschluss wird die Platine auf ihre Funktionalität getestet.

In Kapitel 6 wird das VHDL-Modell erklärt und entwickelt. Anschließend wird das VHDL-Modell auf dem FPGA mit dem Audiocodec-Board evaluiert.

In Kapitel 7 wird als erstes ein FIR-Filter in VHDL entworfen und getestet. Anschließend wird der selbe FIR-Filter mit dem Software Vivado HLS (High Level Synthesis) entworfen und ausgewertet. Zum Schluss wird ein Audioeffekt mit MATLAB entwickelt, dann in Vivado HLS übertragen und schließlich in VHDL synthetisiert und getestet.

In Kapitel 8 erfolgt schließlich die Zusammenfassung der Arbeit.

2. Grundlagen

In diesem Kapitel werden einige Grundlagen behandelt. Diese werden im Laufe der Arbeit zum Tragen kommen. Als Erstes wird allgemein über das menschliche Gehör und die Tontechnik geschrieben. Danach wird der Operationsverstärker und die beiden Grundbetriebsarten Nicht-Invertierend und Invertierend, sowie die Spezielle Filterschaltung Multiple-Feedback erläutert. Danach wird das im Labor genutzte FPGA-Chip und das FPGA-Board beschrieben. Anschließend werden einige Digitale Schaltungen erklärt. Zum Schluss werden Anwendungen zur Signalverarbeitung der FIR-Filter und der Vibrato-Effekt erklärt.

2.1. Bezugspegel in der Tontechnik

In der Tontechnik werden Audiosignalpegel in zwei Arten dargestellt, in dBu und dbV. Die Skalierung ist dieselbe nur der Bezugspunkt ist unterschiedlich. Dem dBu liegt die alte Kommunikationstechnik zur Grunde. Hierbei wird die Spannung an einem 600 Ohm Widerstand die 1 mW effektiv umgesetzt wird als Bezugspunkt gewählt. Beim dBV wird 1 V effektiv als Bezugsspannung gewählt. Die Umrechnung beträgt dBV = 1.29 dBu. In der Anwendung werden verschiedene Pegel für verschiedene Bereiche verwendet. Die Anwendungen werden in zwei Kategorien unterteilt, professionell und Verbraucher Anwendung. Die Pegel bei professionellen Anwendungen sind höher als die der Privatgebrauch. So verwendet das ARD Studio einen Pegel von 6dBu (1.55 V effektiv) und in den Studios in den Vereinigten Staaten 4 dBu (1.228 V effektiv), während im Heimtechnik der Pegel von -10 dBV (0.316 V effektiv) angegeben werden. Der Heimtechnik-Pegel ist jedoch in der Anwendung nicht maßgebend. Beispielsweise werden bei vielen Soundkarten werden eine Spannung von +/-0.7 V angegeben. Jedoch ist das auch nicht der Standard und kann abweichen [19].

2.2. Einfacher Verstärker

Mittels eines Operationsverstärkers kann man einen einfachen Verstärker dimensionieren. Man kann den Operationsverstärker in zwei Grundschaltungen beschalten, in einem Invertierenden und ein Nicht-Invertierenden Schaltung. In dieser Sektion werden beide vorgestellt.

2.2.1. Nicht-Inventierendes Betrieb

Bei der Abbildung 2.1 handelt es sich um einen Nicht-Invertierender Verstärkerschaltung, auch Spannungsfolger genannt.

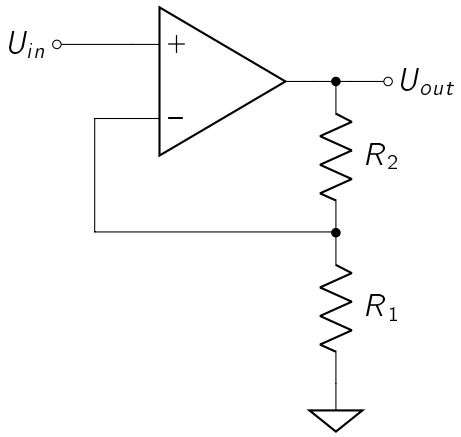


Abbildung 2.1.: Nicht-Invertierende Grundschaltung mit einem Operationsverstärker

Der positive Eingang des Operationsverstärkers wird mit der Eingangsspannung U_{in} verbunden. Die Widerstände R_1 und R_2 befinden sich in Reihe an der Ausgangsspannung U_{out}. Der negative Eingang des Operationsverstärkers wird zwischen den Widerständen verbunden. Der Operationsverstärker verstärkt den Eingangsspannung U_{in} um die Verstärkung G. Allgemein kann man sagen, dass die Verstärkung das Verhältnis von der Ausgangs- zur Eingangsspannung[8]:

$$G = \frac{U_{out}}{U_{in}} \tag{2.1}$$

Durch die Rückkopplung über die Widerstände R_2 und R_1 kann die Gleichung zu[8]

$$G = \frac{R_2 + R_1}{R_1} \tag{2.2}$$

umgestellt werden. Dadurch kann die Verstärkung individuell über die Widerstände verändert werden[8].

2.2.2. Inventierendes Betrieb

In der Abbildung 2.2 ist ein Invertierender Verstärker zusehen. Auch hierbei handelt es sich um einen Operationsverstärker. Das Eingangssignal wird über den Widerstand R_1 an den

negativen Eingang des Operationsverstärkers verbunden. Das Ausgangssignal wird über den Widerstand R_2 an den negativen Eingang des Operationsverstärkers verbunden. Zunächst wird der positive Eingang des Operationsverstärkers mit der Masse verbunden[8]. Die Verstärkung beträgt dann nach der Gleichung 2.1

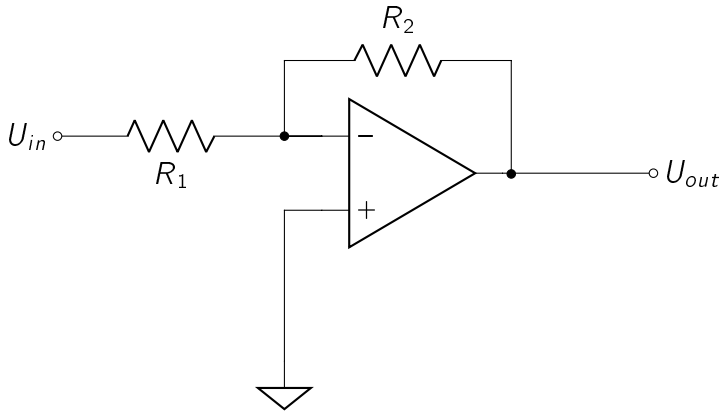


Abbildung 2.2.: Invertierende Grundschaltung mit einem Operationsverstärker

$$G = -\frac{R_2}{R_1} \tag{2.3}$$

Die Verstärkung ist immer negativ zum Eingang[8].

2.2.3. Erweiterte Differenzverstärker

Der Differenzverstärker ist eine Schaltung mit einem Operationsverstärker. Die einfache Differenzverstärkerschaltung besitzt keinen Kondensator C_1. Der Name kommt daher, dass der einfache Verstärker bei gleich gewählten Widerständen eine Ausgangsspannung von der Differenz vom positiven zur negativen Eingangsspannung des Operationsverstärker einstellt[8].

Für die Gesamtverstärkung für den erweiterten Verstärker wird die Verstärkungen für die einzelnen Eingänge separat betrachtet. Dazu wird der andere Eingang gegen Masse geschaltet[8].

Als erstes wird U_{e1} auf Masse gelegt. Man erhält einen Nicht-Invertierender Verstärker mit einem Spannungsteiler am positiven Eingang des Operationsverstärkers. An dem negativen Eingang wird die Ausgangsspannung über den Widerstand R_2 zurück gekoppelt. Zusätzlich ist der negative Eingang mit über den Widerstand R_1 und dem Kondensator C_1 gegen Masse verbunden. Die Verstärkung für diesen Fall beträgt

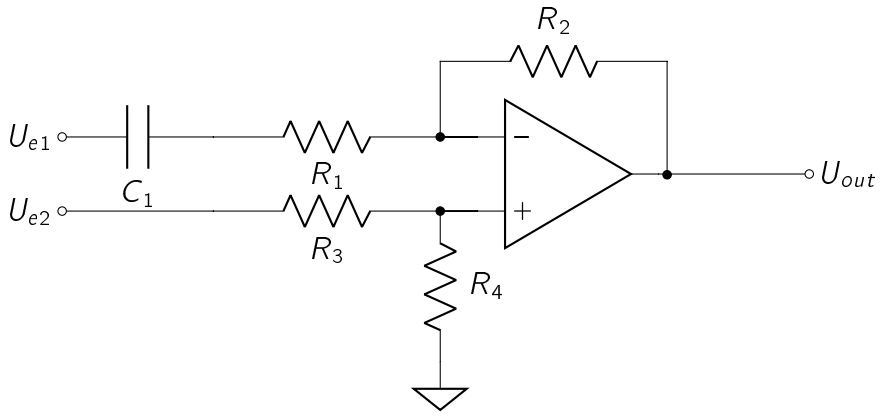


Abbildung 2.3.: Differenzverstärkerschaltung mit einem Operationsverstärker und einem Kondensator am negativen Eingang

$$U_a(j\omega) = \frac{R_4}{R_3 \cdot R_4} \cdot (j\omega C_1(R_1 + R_2) + 1) \cdot \frac{1}{j\omega R_1 C_1 + 1} \cdot U_{e2}(j\omega) \tag{2.4}$$

Durch den Kondensator wird die Verstärkung Frequenzabhängig. Unter Betrachtung der Gleichung 2.4 wird deutlich, dass diese Schaltung ein Hochpass verhalten zeigt. Läuft die Frequenz gegen Null, stellt sich eine Verstärkung von

$$G_{\omega=0} = \frac{R_4}{R_3 + R_4}. \tag{2.5}$$

R4/R3+R4. Steigt die Frequenz gegen Unendlich verhält sich dieser Verstärker wie ein nicht inventierender Verstärker mit dem Verstärkungsfaktor

$$G_{j\to\infty} = \frac{R_4}{R_3 + R_4} + \frac{R_1 + R_2}{R_1}. \tag{2.6}$$

Das System ist somit Stabil und artet nicht bei einer endlichen Verstärkung.

Als nächstes wird U_{e2} auf Masse gelegt. Man erhält ein Invertierender Verstärker. Der positive Eingang liegt über den Widerständen R_3 und R_4 an der Masse. Einfachheitshalber werden diese Widerstände weggelassen. Der Kondensator C_1 und der Widerstand R_1 ist mit dem negativen Eingang verbunden. Der Ausgang ist über R_2 zum negativen Eingang zurück gekoppelt. Die Verstärkung beträgt somit

$$U_a(j\omega) = -j\omega R_2 C_1 \cdot \frac{1}{1 + j\omega R_1 C_1} \cdot U_{e1}(j\omega) \tag{2.7}$$

Anhand der Gleichung 2.7 kann man erkennen, dass sich hierbei wieder um ein Hochpass ähnliches Verhalten vorzeigt. Bei Frequenzen gegen Null stellt sich eine Verstärkung von Eins ein. Wenn die Frequenzen gegen Unendlich steigen, stellt sich die Verstärkung bei -R2/R1 ein. Auch diese Schaltung ist Stabil.

Die gesamte Verstärkung des Differenzverstärkers ist die Summe der Teilverstärkungen

$$U_a(j\omega) = \frac{R_4}{R_3 \cdot R_4} \cdot (j\omega C_1(R_1+R_2)+1) \cdot \frac{1}{j\omega R_1 C_1 + 1} \cdot U_{e2}(j\omega) - j\omega R_2 C_1 \cdot \frac{1}{1 + j\omega R_1 C_1} \cdot U_{e1}(j\omega) \tag{2.8}$$

2.2.4. Multiple Feedback Filter

Beim Multiple Feedback-Filter handelt es sich um einen aktiven Filter zweiter Ordnung. Dazu wird ein Operationsverstärker als ein Integrator geschaltet. Der Filter invertiert das Eingangssignal, also hat dieser Filter eine Phasenverschiebung von 180° [8].

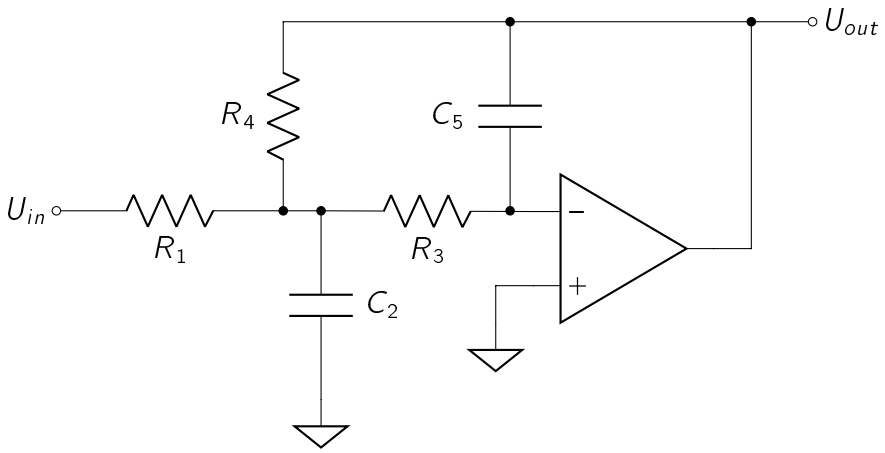


Abbildung 2.4.: Grundschaltung eines Multiple-Feedback-Filters

In der Abbildung 2.4 ist ein Multiple-Feedback-Filter zu sehen. Der Filter besteht aus drei Widerständen und zwei Kondensatoren. Das Eingangssignal ist über R1 und R3 mit dem negativen Eingang des Operationsverstärkers verbunden. Das Ausgangssignal ist über C5 an den negativen Eingang des Operationsverstärkers zurück gekoppelt. Eine zweite Rückkopplung des Ausgangssignals ist über R4. Die zwei Rückkopplungen ist der Namensgeber dieses Filters [8]. Die Übertragungsfunktion für diesen Filter ist

$$\frac{U_{out}}{U_{in}} = \frac{-H\frac{1}{R_1R_2C_2C_5}}{s^2 + s\frac{1}{R_1}(\frac{1}{R_1} + \frac{1}{R_3} + \frac{1}{R_4}) + \frac{1}{R_3R_4C_2C_5}} \tag{2.9}$$

Hierbei ist H der Verstärkungsfaktor. [8]

2.3. ModSys-Baseboard 2

Im Labor wird ein Modulsystem, das ModSys, verwendet. Er besteht aus einem Hauptboard, ModSys Mainboard genannt, sowie verschiedenen Modulen, die über Steckverbinder verbunden werden [9]. Im speziellen wird das ModSys-Baseboard 2 mit dem ARTIX-712 verwendet. Das Mainboard wird über einen externen Netzteil von 5V versorgt. Mittels Spannungsregler wird 3.3V erzeugt, mit der diverse Bausteine mit Spannung versorgt werden, wie der Oszillator und das FPGA Chip, sowie die Bereitstellung an den Steckverbindungen, auch Connector genannt, für die Pheripherimodule. Das Mainboard besitzt einen On-Board Oszillator mit der das FPGA Chip getaktet werden kann. Zudem besitzt das Board die Möglichkeit extern, sowie Manuell getaktet zu werden. Dieser wird über einen Schiebeschalter eingestellt. Das On-Board Oszillator kann Manuell über zwei Taster von einem Hertz bis 80 MHz eingestellt werden.

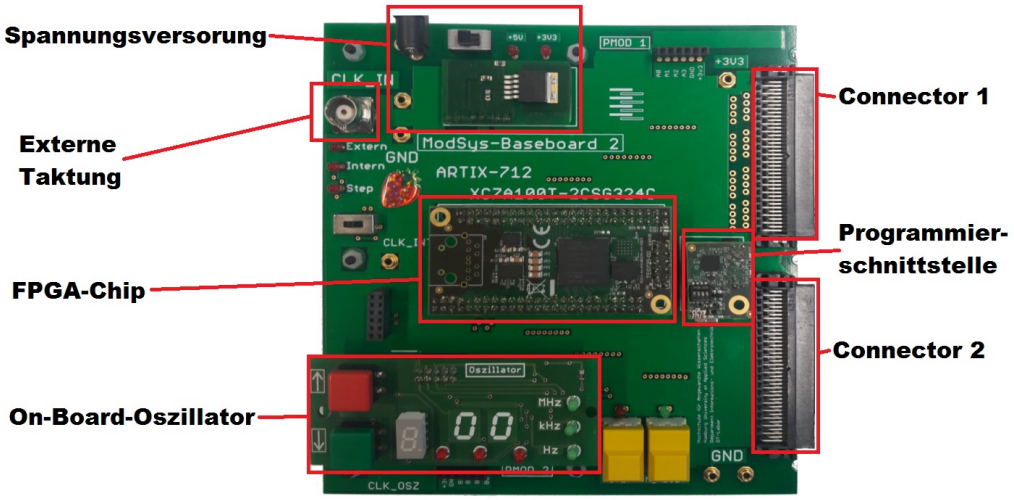


Abbildung 2.5.: ModSys-Baseboard 2

Der FPGA Chip befindet sich auf einer Aufsteckplatine von der Firma "Trenz Electronic". Beim FPAG-Chip handelt es sich um ARTIX-7 XC7A100T mit 210 I/O Pins [10]. Die Auf-

steckplatine hat drei zwei 50er Steckleisten für die I/O Pins sowie eine 12er Steckleiste für den JTAG/UART Schnittstelle [11]. Die Aufsteckplatine wird mit 3.3V versorgt.

Der ModSys-Baseboard 2 verfügt über zwei Connectoren. Jeder Connector besteht aus 80 Pins. An je sechs Pins liegen 3.3V für die Versorgung der Peripherimodule, sowie Masseleitung. Zusätzlich sind 32 Pins des Connectors mit dem FPGA I/O Pins verbunden. Diese 32 Pins wurden zu je 8 Pins gebündelt und somit zu 4 Ports zusammengefasst.

2.4. VHDL Grundschaltungen

In diesem Unterkapitel werden die digitalen Grundschaltungen im FPGA besprochen. Als erstes wird ein digitales Flipflop vorgestellt. Danach wird der Zähler und Taktteiler beschrieben. Anschließend werden die Schieberegister erklärt. Zum Schluss wird die Impulsverkürzung erläutert.

2.4.1. Flipflops

Ein Flipflop ist in der Digitaltechnik ein flanken gesteuertes Speicherelement. Er besteht aus einem Taktsignal C1, einem Eingangsdatensignal 1D und einem Ausgangsdatensignal Q. Nur bei einer vorgegebenen Pegeländerung an C1 wird das Eingangssignal an Q weitergereicht. Hierbei signalisiert die Nummer bei 1D die Abhängigkeit zu C1. Es werden in der Regel auch mehrere Speicherelemente zu einem Speicherblock zusammengefasst und wird Register genannt. Es gibt mehrere Arten von Flipflops wie den RS-Flipflop, während der D-Flipflop die gängigste ist [2].

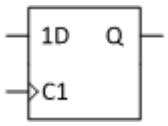


Abbildung 2.6.: Ein einzelner D-Flipflop

In der Abbildung 2.6 sieht man einen D-Flipflop. Der Eingang C1 ist der Takteingang und 1D der Dateneingang und der Ausgang Q ist der Datenausgang. Die Zahl bei C1 gibt die Nummer der Takt an und die Zahl bei 1D zeigt die Abhängigkeit zur Takt C1.

In der Abbildung 2.7 sind drei D-Flipflops zu sehen. Die Eingänge D1 und die Ausgänge Q sind zu einem Bus zusammengefasst. Bei einem Takt wird der Bus gelesen.

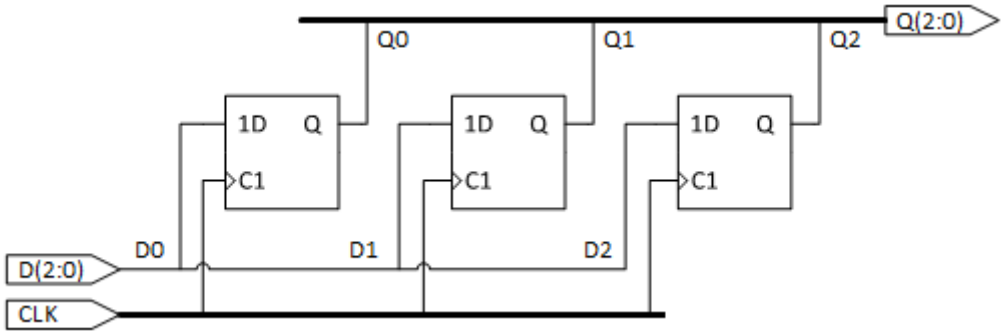


Abbildung 2.7.: Ein Speicherblock von drei D-Flipflops (Register)

2.4.2. Zähler/ Taktteiler

In der Digitaltechnik ist ein Zähler ein Baustein, welcher fortlaufend eine Eins aufaddiert (Vorwärtszähler) oder subtrahiert (Rückwärtszähler) wird. Dabei wird er auch mod-n-Zähler genannt, wobei das n für die Anzahl der Zahlwerte steht. Diese Bezeichnung ist eine Ableitung aus der modulo-Operation der Mathematik, bei der das Ergebnis ein Zahlenwert von 0 bis n-1 ist. Des weiterem wird unter synchronen und asynchronen Zähler unterschieden. Allgemein werden Zähler zum Zählen von Ereignissen und das Herabsetzten einer Taktfrequenz verwendet. Zudem lassen sich durch Zähler Automaten steuern. Im Folgenden wird nur der synchrone Vorwärtszähler beschrieben.

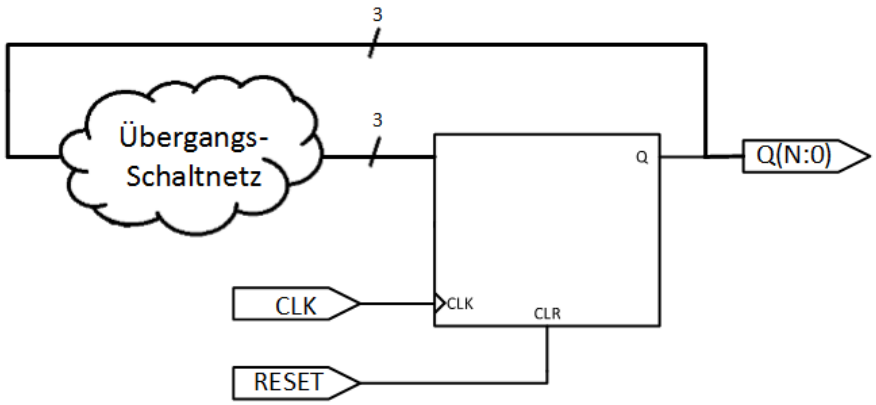


Abbildung 2.8.: Schaltbild eines N-Bit Zählers[2]

Ein Zähler besteht aus einem oder mehreren digitalen Flipflops. Die Anzahl der Flipflops lässt sich durch log2(n) berechnen, wobei das Ergebnis bei einer Fließkommazahl aufgerundet wird. Bei Erreichen der Grenze n-1 wird entsteht ein Überlauf und der Zähler wird zurückgesetzt. Dies geschieht im Übergangsschaltnetz.

Bei der Herabsetzung eines Taktes ist es manchmal wünschenswert, dass der neue Takt synchron zur Taktdomäne ist und dass der neue Takt genauso lange High als auch Low ist (Tastgrad von 50%). Der Tastgrad von 50% wird dadurch erreicht, dass n eine grade Zahl sein muss. Im Ausgangsschaltnetz wird von 0 bis n/2-1 auf Low gesetzt und von n bis n-1 auf High gesetzt.

2.4.3. Schieberegister

In der Digitaltechnik ist ein Schieberegister eine Schaltung von mehreren hintereinandergeschalteten Flipflops. Sie teilen einen gemeinsamen Takt. Mit jedem Takt wird der Register nach links oder auch nach rechts geschoben. Hierbei heißt links Schiebung, dass die Werte des Registers von LSB in Richtung MSB geschoben werden. Rechts Schiebung dahingegen werden die Werte vom MSB in Richtung MSB geschoben.

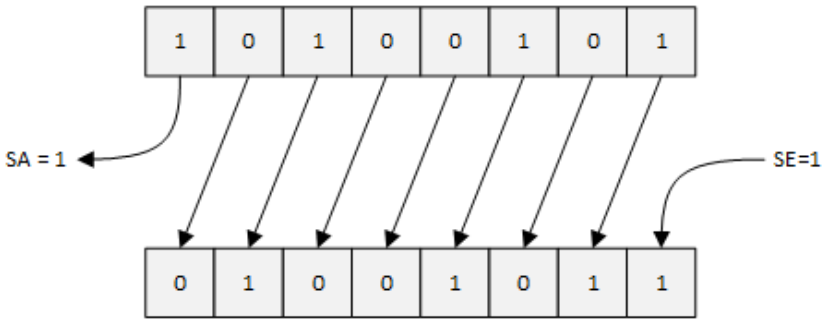


Abbildung 2.9.: Ein linksschiebe Register[2]

Ein Schieberegister ist ein Register, welches sich schieben lässt. Beim Schieben werden die Inhalte der Flipflops an das nachfolgende Flipflop übergeben. Bei einem Linksschieber wird der niedrigste Flipflop mit einem Schiebeeingangsbit SE gefüllt und der höchste Datenbit wird auf den Ausgangsbit verschoben (Abbildung 2.9)[2].

Serien-Parallel-Umsetzer

Ein Serien-Parallel-Umsetzer ist ein Linksschieberegister, dass ein Eingangswert einliest, ihn zu einem Array bildet und das Array gebündelt ausgibt.

Dieser Baustein wird mit dem Systemtakt SYSCLK betrieben. Der EN-Signal wird für die Aktivierung des Linksschiebers benötigt. Das SHIFT-Signal wird für das Schieben der Register benötigt. Ist SHIFT während eines SYSCLK Flanke High, so wird der Schieberegister einmal nach links geschoben und das niedrigste Flipflop wird mit dem Wert von D_IN überschieben.

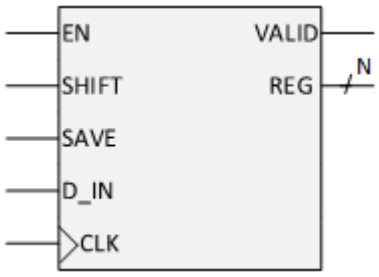


Abbildung 2.10.: Blockschaltbild eines Serien-Parallel-Umsetzers

Beim Anliegen des SAVE-Signals wird bei dem Takt der Schieberegister dem Ausgangsregister übergeben und für eine Taktlänge wird das VALID-Signal auf High geändert.

Parallel-Serien-Umsetzer

Ein Parallel-Serien-Umsetzer ist ein Schieberegister, welcher ein Register einließt und ihn nacheinander ausgibt.

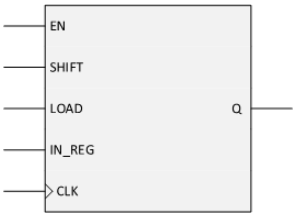


Abbildung 2.11.: Blockschaltbild eines Parallel-Serien-Umsetzers

Der Parallel-Serien-Umsetzer wird mit dem Systemtakt SYSCLK betrieben. Der EN-Signal aktiviert den Schieberegister. Bei einem aktiven LOAD-Signal wird der IN_REG in den Schieberegister eingelesen. Solange SHIFT auf High ist, wird bei jeder steigender Flanke des SYSCLK das höchste Flipflop des Registers an Q übergeben und der Register nach links geschoben. Der Register wird mit Null nachgefüllt.

2.4.4. Impulsverkürzung

Die Impulsverkürzung ist eine digitale Schaltung zur Verkürzung eines Impulssignals auf die Taktzeit. Das Eingangssignal soll auf die Länge eines Systemtakts verkürzt werden. Dazu werden zwei Flipflops aneinandergereiht. Das Ausgangssignal des ersten Flipflops und das negierte Ausgangssignal des zweiten Flipflops werden an einen UND-Gatter geführt.

Das daraus resultierende Signal ist das verkürzte Eingangimpuls. Diese Schaltung wird zur Synchronisation zweier Systeme mit verschiedenen Taktdomänen und zur Detektierung von steigenden Signalflanken verwendet[2].

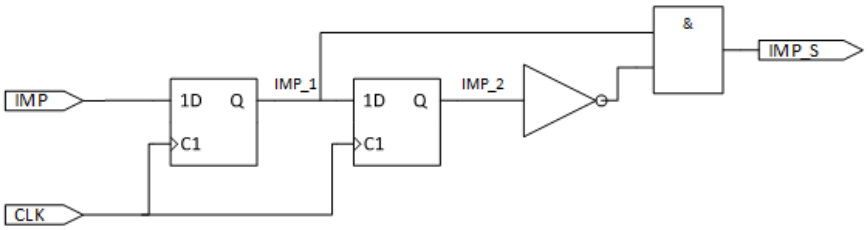


Abbildung 2.12.: Digitale Schaltung einer Impulsverkürzung[2]

Wird IMP1 statt IMP2 negiert und an ein UND-Gatter geführt, wird die fallende Flanke detektiert.

2.5. Digitale Filter und Zeitvariante Verzögerungen

In dieser Sektion wird der FIR-Filter beschrieben, sowie v Vibrato-Effekt beschrieben.

2.5.1. FIR-Filter

Der Finite Impulse Response Filter, kurz FIR-Filter, ist ein digitales Filter, der mit dem aktuellen Eingangssignalabtastwert x[n] und einer bestimmten Anzahl N von vorherigen Eingangswerten x[n-k] den Ausgangswert berechnen lassen kann. Allgemein lässt sich der FIR-Filter im Zeitbereich durch die folgende Differenzialgleichung beschrieben.

$$y[n] = \sum_{k=0}^{N} c_k \cdot x[n-k] \tag{2.10}$$

Zur Bestimmung der Ausgangswerte werden die Eingangswerte mit den Koeffizienten c_k gewichtet. Die Anzahl der zu speichernden Eingangswerte bestimmt die Filterordnung N. Die Filterlänge L gibt die Anzahl der Summanden an (L=N+1) und beschriebt, wie lang die Impulsantwort ist. Da der Filter keine Rückkopplung hat, kann sich der Filter nicht selbstständig zum Schwingung angeregt werden und ist stabil. Die Koeffizienten geben die Art des Filters wieder (Tiefpass, Hochpass) und die Stärke des Dämpfung an[12][4].

Die Gleichung 2.10 wird in den z-Bereich transformiert, um das transiente Verhalten des Filters auszuwerten. Dazu wird die Übertragungsfunktion H(z) ermittelt[12][4].

$$Y(z) = \sum_{k=0}^{N} b_k \cdot z^{-k} X(z) = H(z) \cdot X(z) \tag{2.11}$$

In Gleichung 2.11 ist $z^{-k} = e^{j2\pi kfT}$ die Zeitverschiebungsoperator. Er verursacht eine zeitliche Rechtsverschiebung um kT Abtastintervalle[12][4]. Somit ergibt sich

$$H(z) = b_0 z^0 + b_1 z^{-1} + b_2 z{-}2 + \cdots + b_{N-1} z^{-(N-1)} + b_N z^{-N} \tag{2.12}$$

Aus H(z) ergibt sich das folgende Blockschaltbild

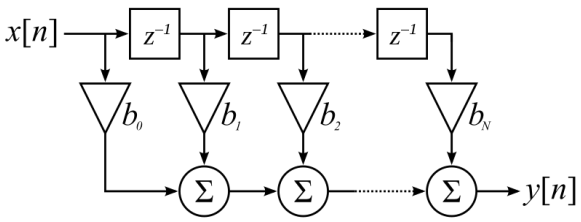


Abbildung 2.13.: Blockschaltbild eines FIR-Filters[3]

Diese Form wird auch die Direktform genannt.

2.5.2. Vibrato-Audioeffekt

Vibrato ist ein Audioeffet basierend auf die Zeitvarianz. Dieser wird erzeugt, indem das Audiosignal mit einer variablen Zeit periodisch verzögert ausgegeben wird. Dadurch wirkt das Ausgangssignal, als ob es schwingen würde. Vibrato wird wie folgt beschrieben

$$y(n) = x(n - M) \tag{2.13}$$

In z-Bereich transformiert ergibt

$$Y(z) = X(z) \cdot z^{-M} \tag{2.14}$$

Der Ausgangssignalwert wird aus dem M-fach verzögerten Eingangssignalwert gebildet. Für den besseren Klang wird ein Hilfsteiler $frac$ eingeführt, der einen Werte zwischen $0 <=$

$frac < 1$ annehmen kann. Dadurch lässt sich die Verzögerung besser einstellen und hat den Effekt, dass der Klang des Audiosignals nicht verzerrt wird[4].

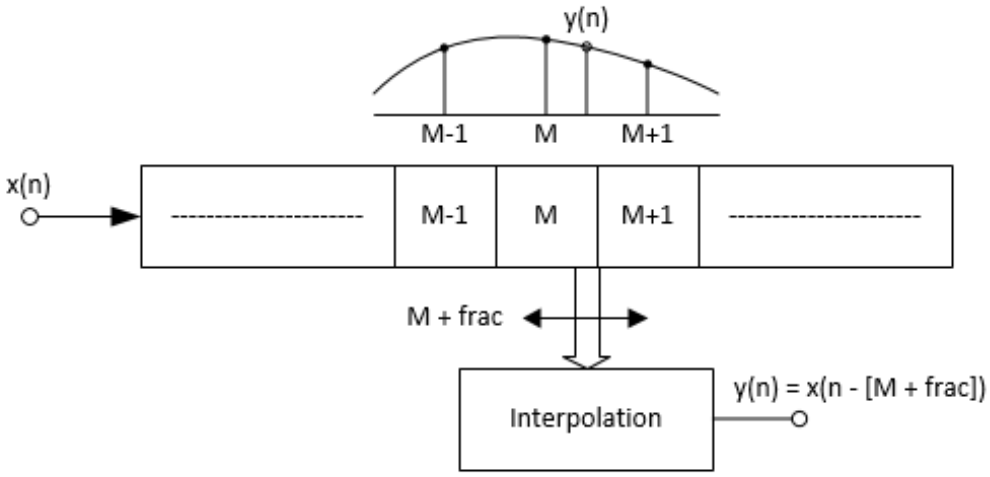


Abbildung 2.14.: Verzögerungskette mit Interpolation[4]

In der Abbildung 2.14 ist eine Verzögerungskette zu sehen, bei der $x(n)$ der Eingangswert und $y(n)$ der Ausgangswert ist. Der neue Eingangswert wird stets an den Anfang gespeichert und die Inhalte der nachfolgenden Werte werden in weiter gereicht. Der letzte Wert fällt aus der Kette raus. Ein Zeiger M zeigt auf die Verzögerungskette. Dieser Zeiger schwingt mit einer niederfrequent über die Verzögerungskette hin und her. Das Kettenglied, auf dem der Zeiger M zeigt, wird an den Ausgang übergeben[4].

Die Position des Zeigers auf die Verzögerungskette wird rechnerisch bestimmt, so zum Beispiel mit Hilfe der Sinuskurve. Die Position des Zeigers liegt meistens zwischen zwei Verzögerungsgliedern. Dieser wird mit dem Hilfsteiler $frac$ erfasst[4].

$$y(n) = x(n - [M + frac]) \tag{2.15}$$

Mittels $frac$ kann der Ausgangssignalwert durch lineare Interpolation bestimmt werden [4].

$$y(n) = x(n - [M + 1]) \cdot frac + x(n - M) \cdot (1 - frac) \tag{2.16}$$

Weitere Interpolationsverfahren können in [4] auf Seite 74 nachgelesen werden.

Dieser Effekt wird oft bei Seiteninstrumenten angewendet um einen vibrierenden Ton zu erzeugen.

3. Anforderungsanalyse

Das Ziel dieses Kapitels ist es, Anforderungen für ein digitales System zur Audioverarbeitung, welches aus einer Platine zur Beschaltung eines Audiocodecs, sowie eines digitalen Systems, implementiert auf einem FPGA, das den Codec digital ansteuert und einen Audioeffekt implementiert. Dieser Audioeffekt soll als Hardware mit Hilfe der High-Level Synthese Software von Xilinx (kurz HLS) implementiert werden.

3.1. Audiocodec

Zu Beginn der Arbeit wurden Anforderungen zur Auswahl des Audiocodecs zusammengetragen.

Der Mensch nimmt akustische Signale in Form von Schallwellen über das Gehör wahr. Dabei kann er Schallwellen mit einer Frequenz von 16 Hz bis ca. 20 kHz wahrnehmen. Tiefe Frequenzen werden als Brummen und hohe als schriller Ton wahrgenommen. Die Sprache liegt im Frequenzbereich 0.5 bis 2 kHz [13]. Der Audiocodec soll mindestens 16 kHz bearbeiten können.

Der Audiocodec soll mindestens einen Kanal, d. h. Mono, bearbeiten können. Jedoch wäre es wünschenswert, wenn er zwei Kanäle, also Stereo, bearbeiten könnte.

Die Konfiguration des Audiocodecs soll für die Studierenden einfach gehalten werden. Die Einstellung des Audiocodecs soll nicht mit einem Mikrocontroller oder dem FPGA erfolgen, stattdessen soll er eine feste Voreinstellung besitzen oder durch eine elektrische Schaltung konfigurierbar sein.

Die serielle Datenübertragung zwischen dem FPGA und dem Audiocodec soll ebenfalls einfach sein. Die serielle Schnittstelle soll standardisiert und geläufig sein, wie z. B. I2C oder I2S.

Die Bit-Auflösung gibt die Audioqualität wieder. Je geringer diese ist, desto verrauschter hört sich anschließend das Audiosignal an. Typische Auflösungen sind 8, 12, 16, 24 und 36 Bits. Die Bit-Auflösung des analogen Audiosignals sollte mindesten 16 Bit betragen.

Die Audiocodec-Bausteine gibt es in diversen Bauformen, so zum Beispiel in Ball Grid Array (BGA, dt. Kugelgitteranordnung), Quad Flat No Leads (QFN), Small Outline (SO) etc. Um Kosten zu sparen, sollten Bauformen, die mehr als zweilagige Platinen erfordern, wie BGA, oder die nur schwer mit der Hand zu löten sind, wie QFN, vermieden werden.

Der Audiocodec sollte vom Hersteller gut dokumentiert sein. Zusätzlich wäre es wünschenswert, wenn schon Evaluationsboards von Hersteller oder andere öffentlich zugängliche Projekte beim Audiocodec auffindbar wären.

Die Anforderung zusammengefasst

- Abtastrate mindestens 32 kHz
- mindestens ein Kanal (Mono), Stereo-Betrieb wäre wünschenswert
- wenig bis keine Konfiguration
- serielle Datenübertragung wie I2C, I2S, PCM, PWM, etc.
- Mindestauflösung 16 Bit
- Bauform gut lötbar
- gute Dokumentation oder bestehende Projekte

3.2. Platine

Die Platine soll als modulare Schnittstelle zum bestehenden Modsys Board benutzt werden. Die Platinengröße muss dementsprechend den anderen Modulen angepasst werden. Passend zur FPGA-Board-Schnittstelle muss das Codecboard eine Kommunikationsschnittstelle haben. Für die zu verarbeitende Audiosignale, die aus einem Computer oder Smartphone kommen, sollten Standardklinkenbuchsen vorhanden sein. Zusätzlich zu diesen sollen die drei Signale linker Kanal, rechter Kanal und Masse über 2mm-Buchsen mit dem Oszilloskop beobachtbar sein. Die Audiosignale sind Line-Level-Pegel. Die Ausgangssignale sollen auch Line-Level-Pegel aufweisen. Auf alle Signale der digitalen Schnittstelle, die für die Kommunikation zwischen dem Audiocodec und dem FPGA erforderlich sind, sollte ebenfalls über 2mm-Buchsen zugreifbar sein. Zudem soll die Audiocodec-Platine Line-Pegel voll auflösen. Des Weiteren bestimmt der ausgewählte Codec weitere Anforderungen sowie die Gestaltung der Leiterbahnplanung. Zusätzlich wird das Buch "Linear Circuit Design Handbook"beim Designen der Platine zu Hilfe genommen.

3.3. VHDL-Modell zur Codecansteuerung

Der FPGA ist das Herzstück der Anwendung, da die Studierenden hier später die Ansteuerung des Audiocodecs und die Signalverarbeitung implementieren sollen. Das Kommunikationsmodul erzeugt die digitalen Ansteuersignale für den Audiocodec und die zeitliche Synchronisation und tauscht außerdem die digitalen Daten mit dem Audiocodec aus. Der Steuerpfad und der Datenpfad sollen im VHDL-Code getrennt werden. Der Code soll Herrn Reicherdts Richtlinien entsprechend geschrieben werden.

3.4. Audioeffekt mit HLS

Im Rahmen des zu entwickelnden Praktikumsversuchs wird angestrebt, dass langfristig die Studierenden mit dem High-Level-Synthese-Tool (HLS) von Xilinx Erfahrungen sammeln und mit dessen Hilfe einen einfachen Audioeffekt generieren. Der HLS ist eine Software, mit der ein VHDL-Code aus einem C/C++ Code synthetisiert werden kann. Anhand eines einfachen Audioeffektes, der auch leicht in VHDL direkt zu implementieren wäre, sollen zunächst Erfahrungen mit der HLS gesammelt werden und diese mit einer direkten VHDL-Implementierung verglichen werden. Mit diesen Erkenntnissen kann im nächsten Schritt ein zu definierender Audioeffekt realisiert werden. Hierbei gilt, dass dieser möglichst eindrucksvoll, aber im Rahmen eines Praktikumsversuches leicht zu implementieren sein soll.

4. Konzeption

In diesem Kapitel werden die einzelnen Konzepte entwickelt und beschrieben. Zunächst werden das Gesamtsystem und seine Komponenten in den Blick genommen. Anschließend wird ein Konzept für den Audiocodec und die Platine des Audiocodecs abgeleitet und entwickelt.

4.1. Das Gesamtsystem

Mit den bekannten Anforderungen aus Kapitel 3 kann ein Konzept für das Gesamtsystem hergestellt werden. Es soll eine Platine entwickelt werden. Dazu gehören der Audiocodec IC und die Zusatzschaltungen für die Stromversorgung sowie die Anpassung der Signalpegel an den Audiocodec.

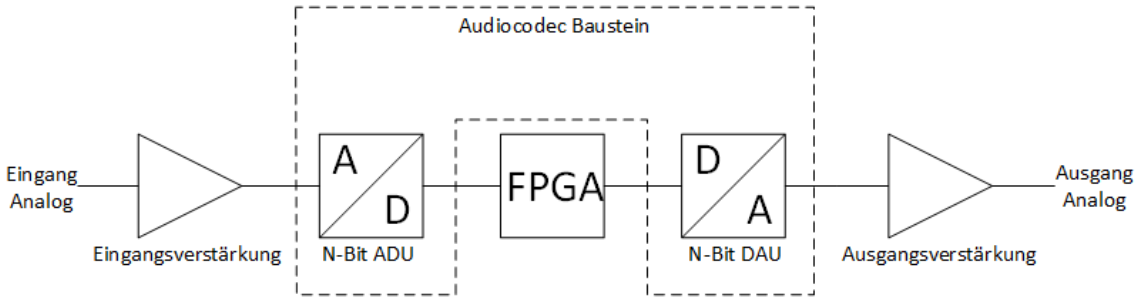


Abbildung 4.1.: Das Signalflussdiagramm für einen Audiokanal

In der Abbildung 4.1 ist das Signalflussdiagramm für das Audiosignal zu sehen. Die analoge Schaltung besteht aus drei Teilen: der analogen Eingangsverstärkerschaltung, dem Audiocodec-Baustein und der Ausgangsverstärkerschaltung. Durch die Schnittstelle wird das Audiocodec-Board mit dem Modsys-Baseboard 2 verbunden.Als erstes wird der Signalpegel des Audiosignals in einer Eingangsverstärkerschaltung an den die Signalpegel des Audiocodec-Bausteins angepasst. Danach wird im Analog-Digital-Umsetzer (ADU) des Audiocodecs das analoge Audiosignal in N-Bits umgewandelt. Alsdann wird das digitale Signal seriell an den FPGA-Chip übertragen. Nach der digitalen Signalverarbeitung im FPGA wird

das N-Bit-Signal wieder zurück an den Audiocodec-Baustein seriell übertragen und mittels eines Digitalen-Analogen-Umsetzers (DAU) wieder in ein analoges Signal umgewandelt. Im Anschluss wird der Signalpegel des analogen Audiosignals wieder zurück angepasst, sodass der Signalpegel wieder mit dem des Eingangssignals übereinstimmt.

4.2. Auswahl des Audiocodecs

Anhand der Anforderungen die im Abschnitt 3.1 hergeleitet worden sind, soll ein Audiocodec-Baustein ausgewählt werden. Zuerst werden die Mindestanforderungen betrachtet, um das Portfolio an Audiocodecs zu minimieren und die Suche einfacher zu gestalten. Die Mindestanforderungen sind jene, die zumindest zu erfüllen gilt. Diese sind

- die Abtastrate, mit dem das System arbeitet, ist mindestens 32 kHz,
- die Bit-Auflösung der Audiocodecs ist mindestens 16 Bit und
- das Package soll mit den im Labor zur Verfügung stehenden Mitteln bestückbar sein (kein BGA oder QFN).

Mit diesen Kriterien wurden aus zahlreichen Audiocodecs jeweils ein passender von vier großen Herstellern ausgewählt. Diese sind Folgende:

- Analog Devices: AD74111,
- Rohm Integrate: BU26156RFS,
- Texas Instruments: PCM3006 und
- ST Microelectronic: STA120 und STA020

Für die oben ausgewählten Audiocodecs wurden die Datenblätter ausgewertet und die spezifischen Daten in der unteren Tabelle zusammengetragen.

Tabelle 4.1.: Ausgewählte Audiocodec-Systeme

	AD7411	BU26156RFS	PCM3006	STA020/120
Abtastrate [kHz]	8-48	4-48	4-48	4-48
Kanäle	1	2	2	1
Bauform	TSSOP	HISSOP	TSSOP	SOP
Konfiguration	Programmierbar	Programmierbar	Stand Alone	Stand Alone
Audiointerface	PCM	PCM	PCM	PCM/I2S
Auflösung [Bit]	16-24	24	16	16-24

Diese Audiocodecs wurden anschließend nach den folgenden Kriterien beurteilt:

- Anzahl der Audiokanäle
- Analoges Interface
- Digitales Interface
- Unterstützung

Alle ausgewählten Audiocodecs können bei einer Abtastrate bis zu 48 kHz betrieben werden, und wie erwünscht haben sie eine Mindestauflösung von 16 Bit. Zudem haben alle die SO (Small Outline) Bauform. Bei SO handelt es sich um eine Surface-Mounted-Device–Bauform (SMD), also eine oberflächenmontierbare Bauform. Die Pins werden flach auf der Oberfläche der Platine, den Pads, gelötet. Die Kommunikation zwischen dem Audiocodec und dem FPGA erfolgt über PCM. Die Konfigurationen müssen bei AD7411 und BU26156RFS über einen Mikrocontroller oder das FPGA eingestellt werden, während bei den anderen beiden die Einstellungen über Spannungspotentiale gesetzt werden können. Alle ausgewählten Audiocodecs können in Mono betrieben werden. BU26156RFS und PCM3006 lassen sich zusätzlich in Stereo betreiben.

Der PCM3006 wurde für die Anwendung als am geeignetsten betrachtet. Außerdem lässt sich die Abtastrate diess Audiocodecs variabel über den Takt einstellen und kann zwei Kanäle verarbeiten. Die Datenübertragung erfolgt über das PCM-Format, und die Auflösung erfolgt auf 16 Bit. Der Audiocodec besitzt Möglichkeiten, die über Pins eingestellt werden können. Dazu wird die Spannung an den Pins entweder auf die Betriebsspannung gezogen oder auf Masse gelegt. Die Bauform des Bausteins ist gut händisch zu löten. Zudem gibt es ein bestehendes Projekt mit diesem Baustein.

Für den Audiocodec spricht weiterhin das einfache Interface. Es können so bereits hardwareseitig die wichtigsten Einstellungen vorgenommen werden und somit der Praktikumsversuch für die späteren Studierenden entlastet werden. Bei den Audiocodecs mit softwarebasierter Konfiguration müssten nach dem Reset Sequenzen von Bitmustern vom FPGA generiert und an den Audiocodec gesendet werden, die bereits zu Anfang des Semesters, in dem die Studierenden den Versuch bearbeiten sollen, Kenntnisse über RAMs und pointerbasierte Zustandsautomaten erfordern. Diese Themen werden jedoch aus didaktischen Gründen erst am Ende des Semesters behandelt. Eine Alternative wäre der Einsatz eines Softcore Mikrocontrollers, der aber noch mehr Knowhow bei den Studierenden erfordert und in der Regel erst auf Niveau der Masterstudiengänge Thema ist.

Die Audioqualität des Audiocodec, z. B. im Hinblick auf Filter und Rauschen (SNR) wurde nicht in die Auswahl einbezogen, da im späteren Praktikumsversuch wegen der einfachen Audioequipments keine besonderen Anforderungen gefragt sind, die nicht bereits ein Standard-Audiocodec erfüllen kann.

4.3. Übersicht über PCM3006 Audiocodec und Schaltungslayout

Im Folgenden werden einige Eigenschaften des PCM3006 vorgestellt, die für das Konzept zum Schaltungslayout die zusätzlichen Anforderungen darstellen.

Allgemein

Der PCM3006 ist ein Audiocodec bestehend aus Analog-Digital-Umsetzer (ADU) und Digital-Analog-Umsetzer (DAU) mit asymmetrischer Signalübertragung. Sowohl der ADU als auch der DAU verwenden für die Umsetzung die Delta-Sigma-Modulation mit 64-facher Überabtastung. Der ADU hat einen Dezimationsfilter und der DAU ein 8-fachen überabtastender Interpolationsfilter. Der DAU hat zusätzlich eine digitale Emphasis-Funktion. Die Abtastfrequenz kann variabel von 4 kHz bis 48 kHz eingestellt und der Betriebstakt variabel in Abhängigkeit von fs auf 256 fs, 384 fs oder 512 fs eingestellt werden. Vier der 24 Pins dienen zur Spannungsversorgung. Der Audiocodec kann in Bereich von 2.7 und 3.6 V betrieben werden[1].

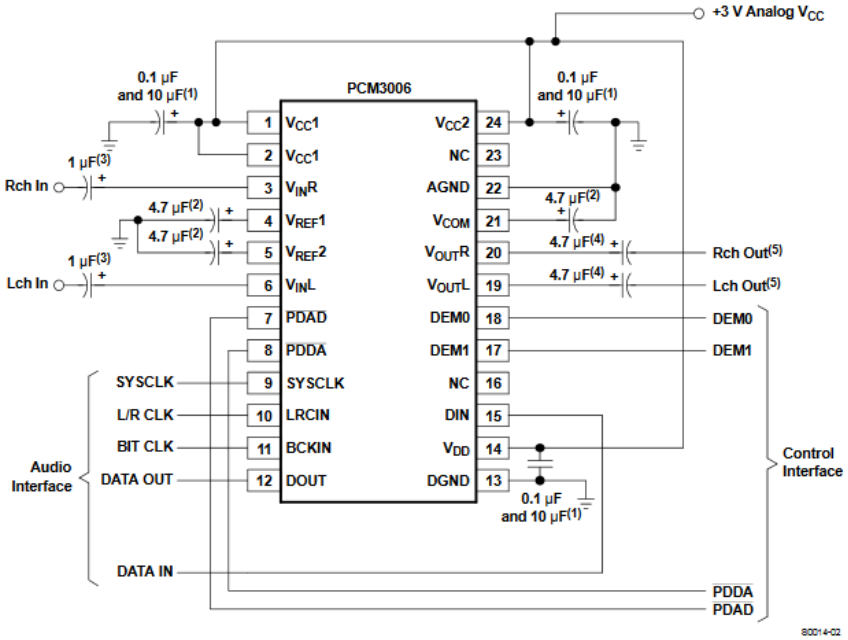


Abbildung 4.2.: Grundschaltung der PCM3006 [1]

Analoge Audioschnittstelle

Die Kanäle des Stereo Audiosignals werden an die Pins $V_{in}R$ und $V_{in}L$ geleitet. Diese werden durch Abblockkondensatoren vom Eingang getrennt. Dadurch werden mögliche Überspannungen am Eingang abgeblockt. Die Pins $V_{in}R$ und $V_{in}L$ haben einen Arbeitsbereich von 0.6 Vcc mit einem Offset von 0.5 Vcc. Folglich heißt es für den Audiocodec, wenn er mit 3.3 V betrieben wird, dass das Audiosignal ein Offset von 0.5 Vcc = 1.65 V haben muss und die maximale Amplitude nicht größer als 0.6 Vcc/2 = 0.99 V sein darf. [1].

Control Interface

Der Control-Interface besteht aus vier Pins $\overline{PDAD}$, $\overline{PDDA}$, $DEM0$ und $DEM1$. Durch das Anlegen eines definierten Spannungspegels an diese Pins lässt sich der PCM3006 konfigurieren. Die definierten Pegel sind zum einen die Betriebsspannung des PCM3006 für den High, sowie die Masse für den Low. Mit den Pins $\overline{PDAD}$ und $\overline{PDDA}$ lassen sich der ADU und der DAU ein- und ausschalten. Diese Pins sind low-aktiv. Die Pins $DEM0$ und $DEM1$ können über den Emphasis-Modus eingestellt werden [1]. Mit der sogenannten Emphasis wird eine Art Rauschunterdrückung beschrieben. Dabei wird das Audiosignal in einem bestimmten Frequenzbereich bei der Aufnahme angehoben und bei der Wiedergabe wieder abgesenkt. Dabei wird darauf geachtet, dass der Frequenzgang linear bleibt [14]. Die Einstellung dieser Control Pins kann sowohl vom FPGA ausgehend erfolgen, als auch durch die Beschaltung an der Platine. Bei der letzteren Variante wäre die Einstellung fix und ließe sich nicht ändern. Bei der ersteren Variante ließe sich der Audiocode individuell einstellen. Um die Ansteuerung des Audiocodec einfach zu halten, wurde die Variante mit der fixen Verschaltung der Pins gewählt. Die Pins $\overline{PDAD}$, $\overline{PDDA}$ sind so eingestellt, dass ADU und DAU nicht ausschaltbar sind. Zudem wird der Emphasis-Modus ausgeschaltet, indem $DEM0$ auf High und $DEM1$ auf Low gesetzt werden [1].

Taktschnittstelle und PCM Dataschnittstelle

Der Audiocodec-Takt (CDCCLK) kann je nach Abtastrate fs des Audiosignals variieren. Dabei muss der Takt das 256-, 384- oder 512-fache von fs haben. Wenn ein CDCCLK mit dem 384- oder 512-fachen von fs benutzt wird, wird der Takt mittels einer integrierten Schaltung in dem PCM3006 auf das 256-fache von fs herunter-geteilt. Die hohe Taktung wird für die internen Filter der Ein- und Ausgangssignale benötigt [1].

Neben CDCCLK benötigt der PCM3006 einen Worttakt (LRCIN) und einen Bittakt (BCKIN). Bei einer steigender Worttakt-Flanke wechselt der PCM3006 zum linken Kanal und bei einer fallenden Worttakt-Flanke zum rechten Kanal. Der Worttakt gibt zudem die Abtastfrequenz

Tabelle 4.2.: Systemtakt Frequenzen [1]

Abtastrate fs [kHz]	Systemtakt		
	256	384	512
32	8.1920	12.2880	16.3840
44.1	11.2896	16.9344	22.5792
48	12.2880	18.4320	24.5760

fs an. BCKIN gibt dem Audiocodec an, wann ein Bit geschrieben bzw. gelesen werden kann. Bei fallender BCKIN-Flanke wird ein Bit geschrieben, bei steigender gelesen. Beim Bitstrom für den DAU handelt es sich um ein rechtsbündiges und beim Bitstrom für den ADU um ein linksbündiges 16-Bit-Datenformat. Der Most Significant Bit (MSB) wird als erstes geschrieben, bzw. gelesen. BCKIN kann 32-, 48- oder 64-fach höher als der Worttakt eingestellt werden (Abbildung 4.3) [1].

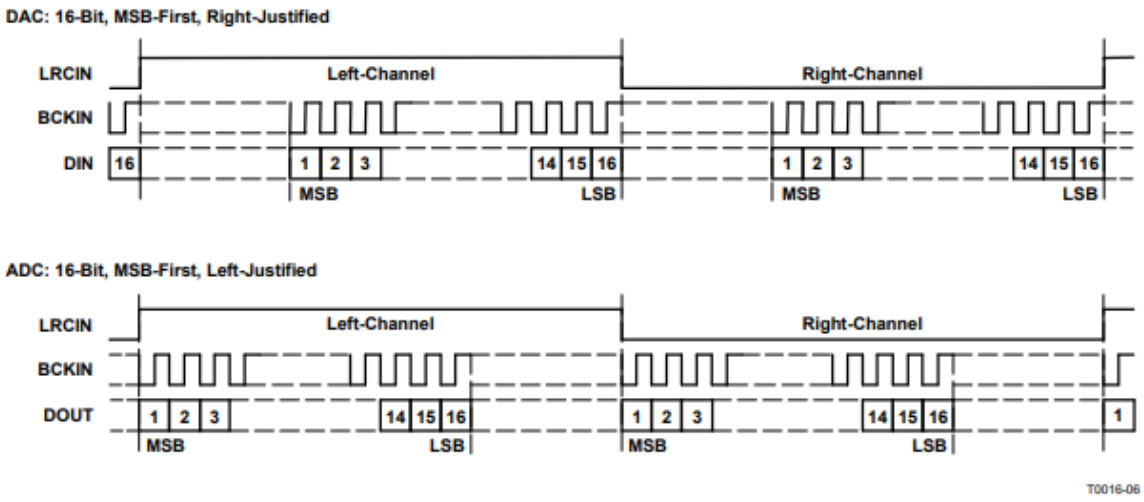


Abbildung 4.3.: Das PCM Signal. Zu sehen ist oben DAC-Signal: Serieller Datenstrom zum Audiocodec; unten ADC-Signal: Serieller Datenstrom vom Audiocodec [1]

Die Takte BCKIN und LRCIN sind jeweils das 2^2- und 2^8-fache vom CDCCLK. Es gibt zwei Möglichkeiten, den Takt zu generieren. Die erste Möglichkeit ist, dass ein Zähler-IC-Baustein z. B. SN74HC590A von Texas Instruments [15], und ein Quarzoszillator auf der Platine eingeplant werden. Der Quarzoszillator mit einem Wert aus der Tabelle 4.2 betreibt den PCM3006. Der Zähler zählt die Takte des Oszillators. Der zweite Pin des Zählers würde das BCKIN-Signal betreiben und der achte Pin das LRCIN-Signal. Die zweite Möglichkeit ist, dass der FPGA alle drei Signale generiert und den PCM3006 ansteuert. Bei der ersten Variante ist der Audiocodec sofort auslesbar, sobald die Platine mit Spannung versorgt wird, jedoch ist die Abtastrate fix und nicht mehr veränderbar. Mit der zweiten Variante ist die Abtastrate fle-

xibel und jederzeit veränderbar, jedoch muss sie zuvor vom FPGA generiert werden. Für die weitere Arbeit wurde die zweite Variante in Betracht gezogen, um die Taktsignale und somit die Abtastrate flexibel zu halten. Zudem entfällt die Synchronisation von BCKIN und LRCIN auf das System, um den Bitstrom DIN/DOUT anzusteuern [1].

Timing der Signale

Der FPGA soll entsprechend der zuvor erläuterten Konzeptionentscheidung die Taktsignale LRCIN, BCKIN und CDCCLK sowie die Datensignale DIN und DOUT generieren. Dabei ist darauf zu achten, dass die vom Hersteller angegebenen zeitlichen Relationen der Taktsignale eingehalten werden. Im Anhang A sind die maximalen sowie die minimalen zeitlichen Verschiebungen der einzelnen Signale zu sehen. Der Takt LRCIN wird als Referenz genommen. Alle weiteren Takte und Signale sind auf ihn referenziert. So muss darauf geachtet werden, dass zum Beispiel nach einer Flanke das neue Datensignal mindestens nach 20 ns anliegen muss.

4.4. Design der Platine

Im Folgenden wird zuerst die Beschaltung des PCM3006-Bausteins beschreiben. Anschließend wird die Pegelanpassung des Audiosignals erklärt.

4.4.1. Grundschaltung

Wie schon erläutert bestimmt der Audiocodec ganz maßgeblich den Schaltungsentwurf für die Platine. Die Spannungsversorgung der analogen ADC- und DAC-Bauteile sowie der digitalen Bauteile wird separat eingespeist. Der ADC wird über die Pins 1 und 2, der DAC über den Pin 24 und die digitalen Bauteile über den Pin 14 versorgt. Dabei ist zu beachten, dass diese Pins mit jeweils zwei Kondensatoren ausgestattet werden müssen, einem 10uF-Elektrolytkondensator und einem 0.1uF-Keramikkondensator. Der Keramikkondensator hat den Vorteil, dass er hochfrequentes Rauschen in der Versorgungsspannung unterdrückt (S.881 in [16]). Der Elektrolytkondensator hingegen liefert als lokaler Ladungsspeicher zusätzlichen Strom für den Fall erhöhten Strombedarfs, sodass die Versorgungsspannung an dem Audiocodec nicht einbricht und verhindert wird, dass Störungen auf das analoge Audiosignal übertragen werden oder sich eine digitale Fehlfunktion ergibt, wenn Vcc die spezifizierten Werte unterschreitet bzw. GND diese überschreitet (S.881 in [16]). Allgemein gilt, dass der Keramikkondensator so nah wie möglich an dem Pin liegen sollte. Der Elektrolytkondensator sollte nicht weiter als 10 cm entfernt vom Pin liegen (S. 87 in [8]). Der PCM3006 hat

asymmetrische Signal-Übertragungseingänge für beide Kanäle. Bei einer asymmetrischen Signalübertragung (engl. single ended) handelt es sich um ein Signal, das sich gegenüber einem Bezugspotential ändert. Dieses liegt in der Regel bei Vcc/2. Ein Kondensator vor dem Eingangspin dient als Gleichspannungsentkopplung und soll die Beeinflussung des Bezugspotentials vor dem Eingangssignal abhalten. Der Wert des Kondensators, der wie ein Hochpassfilter wirkt, soll gerade groß genug sein, um den Gleichanteil zu unterdrücken, aber nicht zu groß, um die niederfrequenten Komponenten des Audiosignals nicht ungünstig zu beeinflussen [17].

Wie oben erläutert wird DEM0 mit der Versorgungsspannung und DEM1 mit der Masse verbunden und die $\overline{PDAD}$- und $\overline{PDDA}$- Pins mit der Masse verbunden. Die Pins für LRCIN, BCKIN, SYSCLK sowie DIN und DOUT werden zur Modsys-Baseboard-Schnittstelle geführt.

4.4.2. Pegelanpassung

Am Eingang des Audiocodecs-Boards kann maximal +/- 1.41 V, d.h. 2.83 V Spitze-Spitze, anliegen. Bei einer Betriebspannung von 3.3 V hat der PCM3006 einen Arbeitsbereich von 1.98V Spitze-Spitze. Aus diesem Grund müssen die Spannungspegel angepasst werden. Dabei ist zu beachten, dass das Eingangssignal gleichanteilsfrei ist, während der Eingang ein Spannungsoffset von 0.5 Vcc erfordert. Zu diesem Zweck wird am Eingang eine Verstärkerschaltung konstruiert und dieser ihm ein Offset verpasst. Mit Hilfe einer Verstärkerschaltung wird der Spannungspegel angepasst. Der Verstärkungsfaktor beträgt der Gleichung zufolge 2.1

$$G_{Eingang} = \frac{0.99V}{1.41V} = 0.7 \tag{4.1}$$

Für die Pegelanpassung wird der Inventierende-Verstärker verwendet, da es mit dem Nicht-Inventierende-Verstärker nicht möglich ist, Verstärkungen kleiner eins zu realisieren (siehe Gleichung 2.2). Um die Gesamtverstärkung des Audiocodec-Boards bei eins zu halten, muss am Ausgang des Audiocodecs noch einmal eine Verstärkung vorgenommen werden. Zusätzlich wird das Ausgangssignal gefiltert, um hohe Frequenzen zu beseitigen. Für die Verstärkung gilt nach Gleichung 2.1.

$$G_{Ausgang} = \frac{1.41V}{0.99V} = 1.43 \tag{4.2}$$

Es sollen Eingangssignale von -1.41 V bis 1.41 V eingelesen werden können, also in einem Betrag von 2.83V. Bei der Auswahl des Operationsverstärkers muss darauf geachtet werden,

dass er die Eigenschaft "Rail-to-Rail"hat. Das bedeutet, dass die Ausgangsspannung sehr nah an die Grenzspannung reichen kann. Während andere Operationsverstärker einen Puffer bis zu einem Volt haben, können diese bis auf 100 mV an ihre Grenzen reichen [16][Seite 21]. Diese Auswahl ist wichtig, da die maximale Versorgungsspannung nur 3.3 V beträgt und das Maximum an Verstärkung herausgeholt werden soll. Für diese Arbeit wurde der LMV358 von On Semiconductor für geeignet empfunden, da diese bis 6V betreiben werden kann und einen Rail-to-Rail-Ausgang hat [18].

4.5. Ansteuerung des Audiocodecs

Nachdem das Konzept für die Platine umrissen wurde, kann das Konzept für die digitale Kommunikationsschnittstelle im FPGA zum Audiocodec diskutiert werden. Das digitale System wird in VHDL beschrieben und modular konzipiert. Die Steuer- und Verarbeitungselemente werden auf kleinere Module aufgeteilt und modular aneinandergesetzt. Das sorgt dafür, dass die Übersicht gewahrt wird und jedes einzelne Modul separat getestet werden kann. Zudem können einzelne Module modifiziert oder ausgetauscht werden.

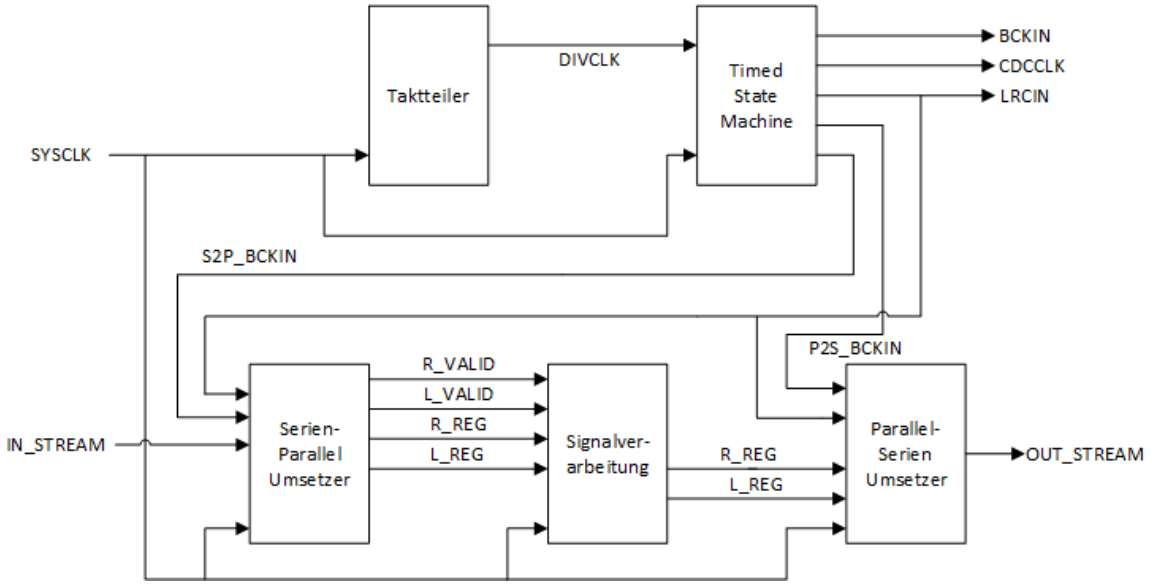


Abbildung 4.4.: Digitale Steuer- und Datenpfade des Kommunikationsmoduls zum Audiocodec

In der Abbildung 4.4 ist die schematische Darstellung der VHDL-Basisstruktur zu sehen. Jeder Block stellt ein Modul dar. Die Blöcke in der oberen Reihe bilden den Steuerpfad und die in der unteren Reihe den Signalpfad. Jeder Block bearbeitet eine Teilaufgabe. CDCCLK

ist der Takt, mit dem der FPGA-Chip betrieben wird. Die Signale BCKIN, CDCCLK und LRCIN werden im FPGA-Chip generiert, um den Audiocodec zu steuern. Das Eingangssignal IN_STREAM beinhaltet die digitalen Datenwerte. Das Ausgangssignal OUT_STREAM wird an den Audiocodec übergeben. Im Steuerpfad befinden sich folgende Blöcke:

- Taktteiler: Der Systemtakt wird auf den Audiocodectakt herunter getaktet.
- Preclock Divider: Der Systemtakt wird auf den Audiocodectakt herunter-getaktet.
- Timed-State-Machine: Die Steuersignale für die einzelnen Module sowie für den Audiocodec werden hier generiert.

Im Signalpfad befinden sich die folgenden Blöcke

- Serien-Parallel-Umsetzer-Modul: Der serielle Datenstrom des Audiocodecs, in dem linker und rechter Audiokanal gemultiplext übertragen werden, wird in separate 16-Bit-Datenwörter für den linken und rechten Kanal umgewandelt.
- Das Signalverarbeitungsmodul: Die 16-Bit-Eingangswerte werden hier digital verarbeitet.
- Ein Parallel-Serien-Umsetzer-Modul: Die 16-Bit-Datenwörter von linkem und rechtem Audiokanal werden in einen seriellen Datenstrom umgewandelt.

Im nachfolgenden Unterkapitel werden die Module näher erklärt.

4.5.1. Serien-Parallel-Umsetzer-Modul

Der serielle Datenstrom vom PCM3006 muss für die weitere Signalverarbeitung je Kanal in ein 16-Bit Datenwort umgesetzt werden. Da zuerst das MSB gesendet wird, wird ein Links-Schieberegister verwendet. Nach einer erfolgreichen Umsetzung wird ein VALID-Signal gesetzt, um dem nachfolgenden Modul mitzuteilen, dass ein neuer gültiger Wert anliegt. Dieses Modul kann im Grundtakt des FPGAs (SYSCLK) sowie des Audiocodecs (CDCCLK) betrieben werden. Für die weitere Bearbeitung wird dieses Modul im SYSCLK getaktet. Dadurch lässt sich das Audiosignal im Modul schneller verarbeiten und das VALID-Signal schneller ausgeben.

4.5.2. Signalverarbeitungsmodul

In diesem Modul wird das 16-Bit-Stereoeingangssignal verarbeitet und anschließend als ebensolches ausgegeben. Für den Einstieg wird das Eingangssignal direkt an den Ausgang übergeben. So soll die grundlegende Funktionalität überprüft werden. Wenn das gewährleistet ist, können weitere Manipulationen in diesem Modul erstellt werden. Das weitere Vorgehen folgt im nächsten Kapitel.

4.5.3. Parallel-Serien-Umsetzer-Modul

Die 16-Bit-Datenworte müssen für die Ausgabe am PCM3006 wieder zurück in einen seriellen Datenstrom von 16-Bit umgesetzt werden. Das 16-Bit-Datenstrom soll dem Audiocodec beginnend mit dem MSB seriell gesendet werden. Dazu wird ein Links-Schieberegister verwendet. Zum Takten wird hier auch der schnelle Takt SYSCLK verwendet. Das Valid-Signal wird nicht benötigt, da das Modul kein Folgemodul hat.

4.5.4. Taktteiler-Modul

Der FPGA kann mit dem On-Board-Oszillator mit bis zu 80 MHz getaktet werden. Da der Audiocodec hingegen mit 12.288 MHz arbeitet, muss der Takt entsprechend geteilt werden. Es bestehen zwei Möglichkeiten, den Systemtakt (der Takt, mit dem das FPGA getaktet wird) einzustellen. Die erste Methode ist, den Grundtakt so einzustellen, dass er mit dem Takt, mit dem der Audiocodec getaktet werden soll, übereinstimmt. Dadurch ist es nicht nötig, einen geeigneten Taktteiler zu finden und einzustellen. Jedoch wird dadurch der FPGA langsamer, und die Anzahl der Takte zwischen zwei Werten ist gering. Bei der zweiten Methode wird mit zwei Taktdomains getaktet, zum einen mit 80 MHz für den FPGA-interne Taktung und zum anderen mit 12.288 MHz für die Kommunikation mit dem Audiocodec. Hierbei kann der Grundtakt maximal gehalten werden, während ein zweiter geringerer Takt den Audiocodec steuert. Dadurch werden längere Bearbeitungszeiten möglich. Jedoch muss ein geeigneter Taktteiler bestimmt und generiert werden. Für die Weiterbearbeitung wurde die zweite Methode gewählt, da die Verarbeitungszeit maximiert werden soll. Das erzeugte Taktsignal wird DIVCLK genannt. Der Teiler muss eine gerade ganze Zahl sein, damit die nachfolgenden Signale richtig generiert werden können.

4.5.5. Timed-State-Machine

Das Timed-State-Machine-Modul hat zwei Aufgaben. Zum einen soll es die Taktsignale LRCIN, BCKIN und CDCCLK zum Steuern des Audiocodecs generieren, und zum anderen soll es die Module im Datenpfad ansteuern. Die Taktsignale LRCIN, BCKIN und CDCCLK werden mit Hilfe von Zählern generiert. Der Zähler inkrementiert bei jeder steigenden Flanke von DIVCLK um Eins. Die Steuersignale für das Serien-Parallel und Parallel-Serien-Umsetzer-Modul können auf zwei Arten erzeugt werden. Die erste Methode ist das Arbeiten mit einer Finite State Machine (FSM). So kann man beispielsweise für jedes Modul ein FSM schreiben. Das Seriell-Parallel-Modul und das Parallel-Seriell-Modul bestehen jeweils aus vier Zuständen (Abbildung 4.4.

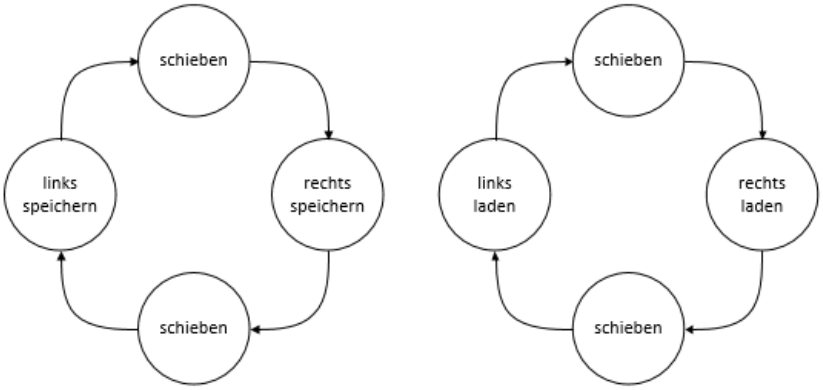


Abbildung 4.5.: Zustandsautomaten zum Steuern der Schieberegister; Links zum Speichern, recht zum Laden

In der Abbildung 4.5 sind zwei Zustandsautomaten zu sehen. Der linke Automat steuert den Serien-Parallel-Umsetzer (SPU) und der rechte Automat den Parallel-Serien-Umsetzer (PSU). Der linke Automat schiebt 16 Bits und lässt anschließend das rechte Schieberegister in einem rechten Zwischenregister speichern. Danach schiebt der linke Automat weitere 16 Bits und lässt danach das linke Schieberegister in den linken Automaten speichern. Analog zu dem linken Automaten wird der rechte Automat zum Laden der Zwischenregister in die Schieberegister verwendet.

Bei der zweiten Methode werden die Steuersignale mit Hilfe von BCKIN und LRCIN erzeugt. Das Schieben des SPU bzw. PSU soll bei einem Pegelwechsel von BCKIN von High zu Low, bzw. von Low zu High erfolgen. Mit Hilfe der Impulsverkürzung können diese Pegeländerungen ermittelt werden. Dasselbe gilt auch für die Signale zum Speichern der Werte von SPU und das Laden der Werte in PSU.

Für die Weiterbearbeitung wurde die zweite Methode gewählt. Der Grund hierfür ist, dass

zum einen mit einer hohen Frequenz getaktet wird. Zum anderen entfällt das Erstellen der Automaten.

4.6. Konzept der Entwicklung eines Audioeffektes und High-Level Synthese

Das Signalverarbeitungsmodul soll entwickelt werden. Für den ersten Test werden die Eingangsregister direkt an die Ausgangsregister übertragen. Für das weitere Vorgehen wird ein simpler Effekt verwendet. Der Effekt soll im diskreten Zeitbereich sein.

Dafür wurde ein FIR-Filter gewählt. Der FIR-Filter besteht aus mehreren kaskadierten Registern, die mit dem Eingangssignal multipliziert und anschließend aufsummiert und ausgegeben werden. Dafür wird ein VHDL-Code geschrieben und getestet. Hierbei wird für die automatische Generierung der Addierstufen das `for loop`-Konstrukt verwendet. Es gibt zwei gängige Varianten für eine Realisierung eines FIR-Filters. Die erste Variante ist eine Additionskette. Dabei werden die Produkte P[k] nacheinander aufsummiert und ausgegeben. Dadurch werden bei N Produkten N-1 Additionen berechnet, welche sequenziell berechnet werden müssen. Bei der zweiten Variante handelt es sich um einen balancierten Addierbaum. Dabei werden jeweils paarweise Produkte P[k] addiert. Anschließend werden die Summen wieder paarweise addiert, bis die Gesamtsumme gebildet wurde. Durch die parallele Addition wird Zeit eingespart. Die zweite Variante wird für die weitere Arbeit verwendet.

Das High-Level-Synthese-Tool (HLS) soll getestet werden. Der Hersteller stellt hierfür einige Anleitungen und Beispiele zur Verfügung. Anhand eines FIR-Filter-Beispiels wird dieses Werkzeug getestet. Mit dem HLS-Werkzeug kann der C/C++ Code mit verschiedenen Einstellungen synthetisiert werden. Dazu werden einige Einstellmöglichkeiten miteinander verglichen. Anschließend wird ein weiterer Audioeffekt mit HLS implementiert. Dieser Effekt soll möglichst einfach zu implementieren sein. Zudem soll die Signalverarbeitung im Zeitbereich stattfinden. Ausgewählt wurde der Vibrato-Effekt. Bei Vibrato handelt es sich um einen auf Zeitverzögerung basierenden Audioeffekt.

5. Schaltungs- und Platinenentwurf

In diesem Kapitel wird die elektrische Schaltung entworfen und erklärt. Die Schaltung besteht aus der Audiocodec Schaltung, einer Ein- und Ausgangsverstärkungsschaltung der analogen Audioein- und ausgänge sowie der digitalen Schnittstelle zum FPGA-Board. Mittels der Eagle-Software wird die Schematics entworfen und aus dieser das Boardlayout erstellt und entworfen. Anschließend wird das Boardlayout extern bestellt. Nach dem Erhalt der Platine wird bestückt. Im Anschluss wird die Platine getestet, indem die Spannungen überprüft werden.

5.1. Schaltungsentwurf

Die Gesamtschaltung aus dem Kapitel Konzeption wird hier entworfen. Zunächst wird die Verstärkerschaltung zur Pegelanpassung am Eingang und dann am Ausgang nach 4.4.2 entworfen.

Nach der Gleichung 2.1 beträgt die Verstärkung bei einer Eingangsspannung von 1.414 V und einer gewünschten Ausgangsspannung von 0.99 V etwa 0.7. Das entspricht etwa -3 dB.

Wie in Kapitel 4.4.2 beschrieben, werden die Pegel mit einem Invertierenden-Verstärker mit dem Faktor 0.7 verstärkt. Mit Hilfe von 2.3 Widerständen lässt sich die Verstärkung mittels der Gleichung 2.3 einstellen. Dazu wird ein Widerstand frei gewählt und der andere an ihn angepasst. Bei der Größe des Werts wurde darauf geachtet, dass der Strom in der Schaltung nicht zu groß wird. Zur Stromverträglichkeit an den Audiosignalein- und ausgängen steht im Datenblatt nichts[1]. In Anbetracht der Stromverträglichkeiten der digitalen Ein- und Ausgänge von 1 bis 100 uA wurde für die Dimensionierung der Widerstände eine angemessene Stromverträglichkeit in der gleichen Größenordnung in uA angenommen. So wurde für die Widerstände R2=39 kOhm und R1=27 kOhm gewählt. Der Kondensator hat den Wert 10 uF.

Mit LTspice lassen sich Schaltungen modellieren und simulieren. Es besteht die Möglichkeit, eine entworfene Schaltung unter anderem zeit- und frequenzabhängig zu simulieren. Mit LTspice wurde die oben beschrieben Schaltung aufgebaut und frequenzabhängig simuliert.

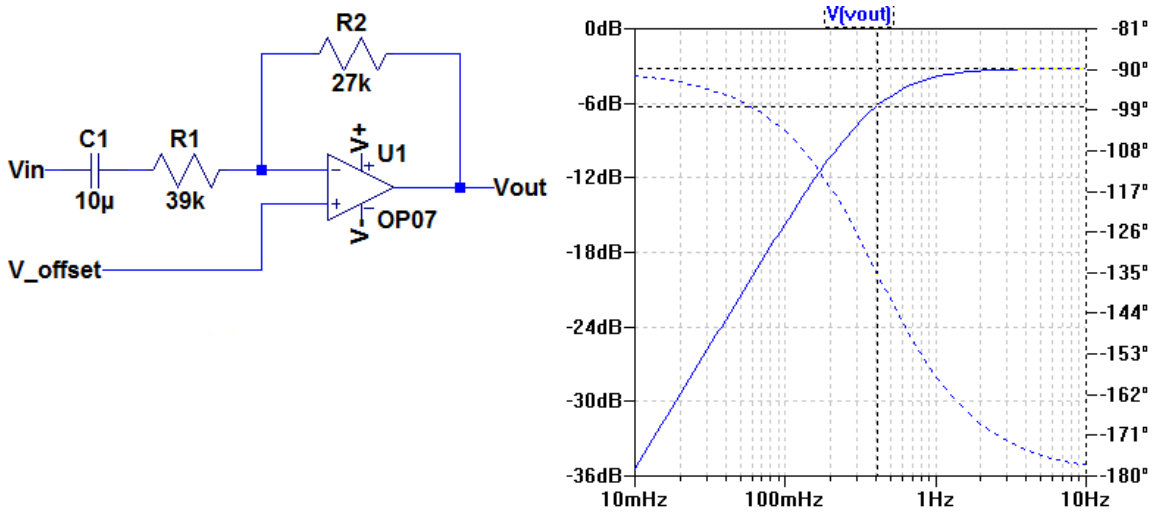


Abbildung 5.1.: LTspice Simulation der Eingangsschaltung; Links: Verstärkerschaltung, Rechts: Amplituden und Phasengang der Schaltung im Bereich 10mHz bis 10 Hz

In der Abbildung 5.1 sind links die Schaltung und rechts der Amplitudengang zu sehen. Im Amplitudengang ist zu erkennen, dass die Grenzfrequenz -3 dB etwa bei 400 mHz liegt. Diese Frequenz ist für das menschliche Gehör nicht mehr wahrnehmbar und somit für die Signalverarbeitung von Audiosignalen nicht relevant[19].

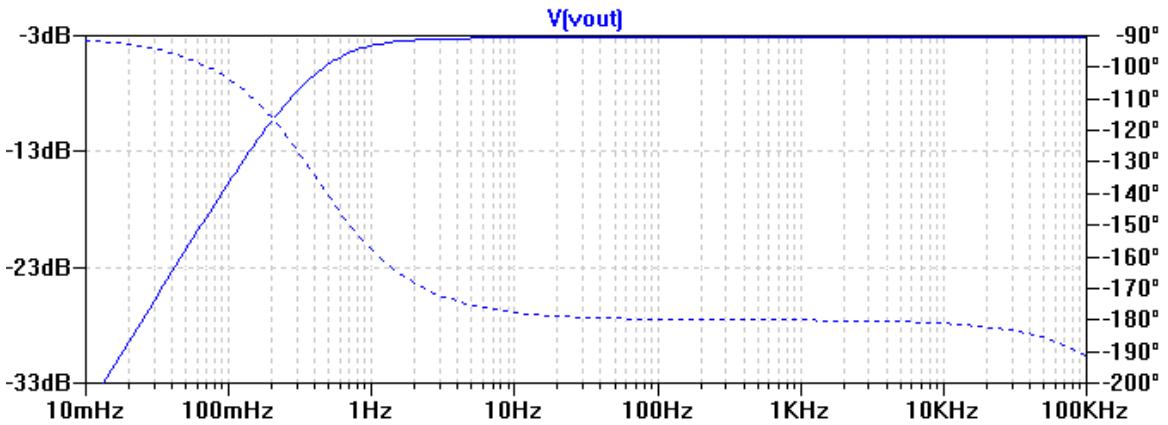


Abbildung 5.2.: Amplituden- und Phasengang der Eingangsverstärkerschaltung in Bereich von 10mHz bis 100kHz

In der erweiterten Sicht des Amplituden- und Phasengangs ist im hörbaren Bereich von 16 Hz bis 20 kHz eine konstante Verstärkung von etwa -3 dB zu sehen. Dieser Wert entspricht etwa einer Amplitudenverstärkung von 0.7, und somit stimmt er mit dem in Kapitel 4.4.2 erweiterten Wert überein. Zudem ist im Phasengang die Invertierung des Wertes zu sehen.

Im Arbeitsbereich ist die Phasenverschiebung fast konstant bei -180 °. Im Bereich von 16 Hz bis 20 kHz ist eine leichte Veränderung zu sehen, jedoch ist dieser sehr gering und somit vernachlässigbar.

Die Ausgangsschaltung soll wie oben beschrieben eine Verstärkung von 1.43 (3 dB) haben und mit einem Multiple-Feedback-Tiefpass realisiert werden. Die Grenzfrequenz soll bei 28 kHz liegen. Dieser Wert kommt dadurch zustande, dass das Audiocodec-Board Frequenzen bis zu 24 kHz bearbeiten soll. Der Spielraum von 4 kHz wurde gewählt, da das Nutzsignal möglichst unverändert bei 24 kHz bleiben soll. Bei der Größe der Werte wurde auch hier darauf geachtet, dass ein angemessener Strom von wenigen 10 uA fließt. Daraus folgen die Werte für die passiven Bauteile: R1 = 6.2 kOhm, C2 = 2700 pF, R3 = 3.9 kOhm, R4 = 10 kOhm, C5 = 300 pF. Bei einer Ausgangsbelastung von 10 kOhm ergibt sich für die Simulation mit LTspice folgender Frequenzgang:

In Bild 5.3 ist ein Tief- und Hochpassverhalten zu sehen. Wie erwartet erzeugt der MF-Filter einen Tiefpass mit einer Grenzfrequenz bei etwa 20 kHz. Das Hochpassverhalten wird vom Koppelkondensator C6 am Ausgang des Operationsverstärkers erzeugt. Die Verstärkung in dieser Schaltung beträgt etwa 3 dB. In den Grenzen von 20Hz bis 20kHz bleibt die Verstärkung konstant. Von 10 kHz bis 20 kHz steigt die Verstärkung um wenige dB.

Die Offset-Spannung wird mit einem Spannungsteiler von Widerständen realisiert. Die Werte wurden R1 = R2 = 10 kOhm gewählt. Dadurch wird der Strom auf einige 100 uA begrenzt. Zusätzlich wird ein Elektrolytkondensator parallel zu R2 angebracht, um mögliche Spannungseinbrüche zu kompensieren.

5.2. Platinenentwurf

Die Platine wurde mit Eagle entworfen. Eagle ist eine Software, mit der sich PCB-Platinen (Printed Circuit Boards) designen lassen. Ein Eagle-Projekt besteht aus zwei Ansichten, der Schaltplan und der Board-Ansicht. Auf der Schaltplan-Ansicht werden aus mehreren Bibliotheken die Bauteile zusammengetragen. Jedes Bauteil hat zwei Formen, eine schematische Darstellungsform für den Schaltplan und eine reale Bauform für die Board-Ansicht. Manche Bauteile oder Bauformen existieren nicht in den vorgegebenen Bibliotheken. Diese müssen dann eigenständig erstellt werden. Auf der Schaltplan-Ansicht werden die einzelnen Bausteine zu einer Schaltung verdrahtet. Anschließend werden in der Board-Ansicht die Bauteile platziert und geroutet. Für die Audiocodec-Platine wurde die Version 7.3.0 verwendet. Im Anhang B.1 ist der gesamte Schaltplan zu sehen.

Die Breite der Traces wurde schmal gehalten, da die fließenden Ströme nicht hoch sind. Die restliche Fläche wurde als Massefläche bestimmt. Dabei wurde die Massefläche in eine analoge und eine digitale Fläche aufgeteilt. Dadurch wird verhindert, dass Störungen auf

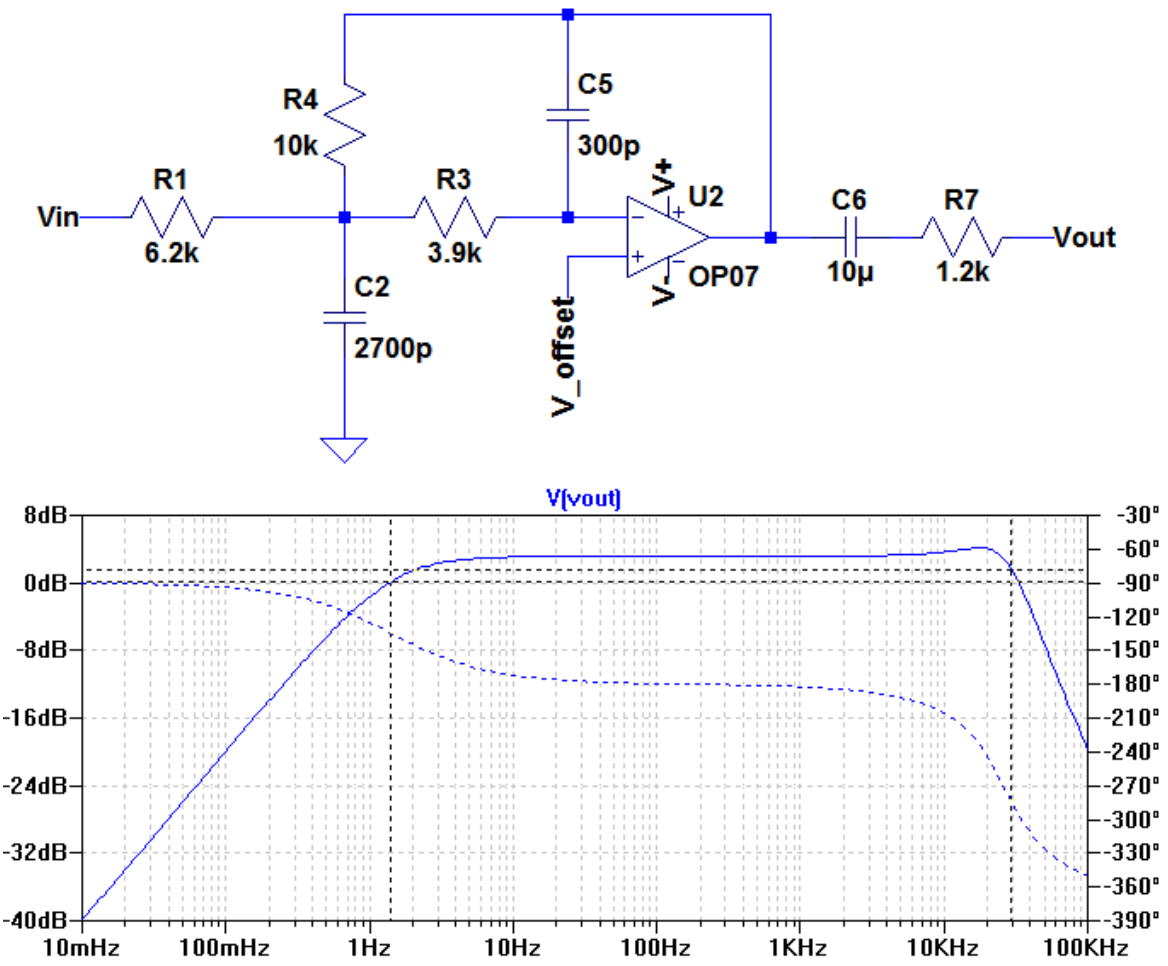


Abbildung 5.3.: LTspice Simulation der Ausgangsverstärkerschaltung; oben: die Multiple-Feedback-Filterschaltung, unten: Amplituden- und Phasengang der Multiple-Feedback-Filterschaltung in den Grenzen von 10mHz und 100kHz

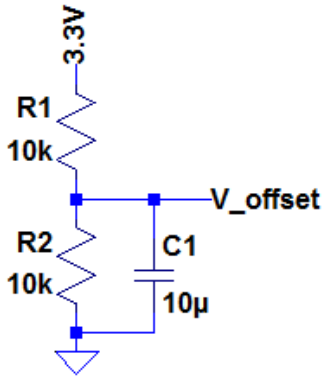


Abbildung 5.4.: Spannungsteiler als Offset-Schaltung

die andere Seite überlaufen. Die Masseflächen werden durch eine schmale Tracebrücke miteinander verbunden[8].

Nachdem sie bestellt und bestückt wurde, sieht die fertige Platine wie folgt aus:

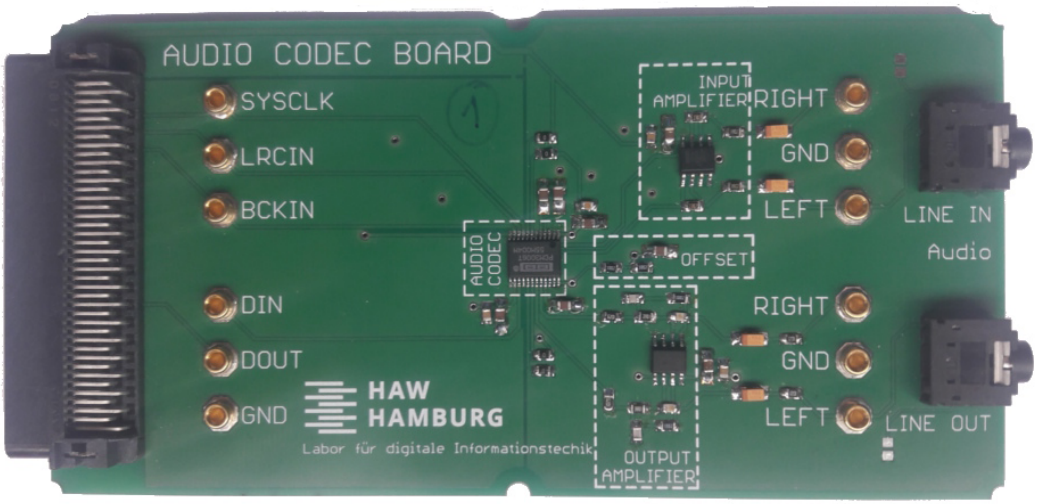


Abbildung 5.5.: Bedruckte und bestückte Audiocodec-Platine

In Abbildung 5.5 ist die fertige Audiocodec-Platine mit den Eingangs- und Ausgangsfilterschaltungen sowie der Offset-Schaltung zu sehen.

5.3. Evaluation der Platine

In diesem Unterkapitel wird die Evaluierung der Platine besprochen. Dazu gehört das Testen der Spannungen. Das Audiocodec-Board wird dafür an das Modsys-Baseboard 2 angeschlossen. Dadurch wird das Audiocodec-Board mit einer konstanten Spannung von 3.3 V versorgt. Da auch der On-Board-Oszillator auf dem ModSys-Baseboard 2 liegt, können auf der Versorgungsspannung nicht erwünschte Oszillationen auftreten. Die Versorgungsspannung an dem PCM3006 sowie an den Operationsverstärker werden auf die konstante Spannung überprüft. Mit Hilfe eines Oszilloskops wurde die Versorgungsspannung untersucht. Die Gesamtverstärkung wird im nachfolgenden Kapitel erörtert.

6. Digitale Schaltungsdesign

In diesem Kapitel wird die digitale Schaltung für VHDL zur Ansteuerung des PCM3006 beschrieben. Dazu werden die einzelnen Module, die in dem Kapitel Konzeption beschrieben wurden, näher erklärt. Die VHDL-Codes wurden mit Hilfe von ModelSim geschrieben und getestet. ModelSim ist eine Entwicklungsumgebung von Mentor Graphics. Mit ihr können VHDL-Codes überprüft und simuliert werden. In dieser Arbeit wurde mit der Studenten-Edition 10.a gearbeitet.

Anschließend werden die digitalen Schaltungen der einzelnen Module zu einem Basiscode zusammengefasst. Zuerst wird der Basiscode vom ModelSim getestet und anschließend mit Vivado Design Suite analysiert. Vivado Design Suite, im folgenden nur als Vivado bezeichnet, ist eine Entwicklungsumgebung von Xilinx für die Synthese und Analyse von Hardwarebeschreibungssprachen. In dieser Arbeit wurde die Vivado-Version 2017.2 verwendet.

Für die Simulation mit ModelSim wurde ein Systemtakt von 100 MHz gewählt und mit Vivado für die spätere Ausführung mit dem Audiocodec Board ein Systemtakt von 80 MHz verwendet. Der Grund für die unterschiedliche Auswahl der Systemtakte liegt darin, dass die Simulation mit ModelSim mit 80 MHz nicht unterstützt wird. Die Auswahl eines Systemtakts von 80 MHz beruht darauf, dass auf dem On-Board-Oszillator maximal dieser Wert eingestellt werden kann.

Die einzelnen VHDL-Codes für die Module sowie des Gesamtsystems befinden sich im Anhang C.

6.1. Taktteiler-Modul

In diesem Modul wird aus dem SYSCLK-Takt eine weitere Taktdomain generiert. Dazu muss ein geeigneter Taktteiler ermittelt werden. Dieser wird anhand des Systemtakts und des maximalen CDCCLK-Takts berechnet.

In der Abbildung 6.1 ist das Blockschaltbild des Taktteiler-Moduls zu sehen. Als Eingangssignal wird lediglich SYSCLK angelegt, und das Ausgangssignal ist DIVCLK. Dazu wird der Systemtakt durch das Doppelte des maximalen Betriebstakts des Audiocodecs dividiert. Der

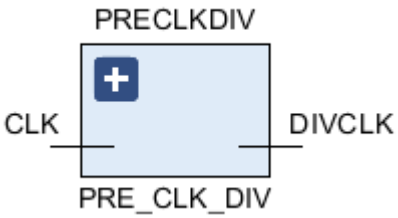


Abbildung 6.1.: Blockschaltbild des Taktteiler-Moduls

Grund für die doppelte CDCCLK liegt darin, dass bei steigender Flanke von DIVCLK ein Pegelwechsel von CDCCLK erzeugt werden soll.

Bei einem Systemtakt von 80MHz und dem maximalen Betriebstakt des Audiocodecs von 12.288 MHz ergibt

$$\frac{80MHz}{2 \cdot 12.288MHz} = 3.25 \tag{6.1}$$

In der Digitaltechnik ist es nur bedingt möglich, einen Takt durch eine Gleitkommazahl zu teilen. Aus diesem Grund wird auf die nächst größere ganze Zahl aufgerundet, in diesem Fall auf 4. Mit Hilfe eines mod-4-Zählers wird die neue Taktdomäne erzeugt.

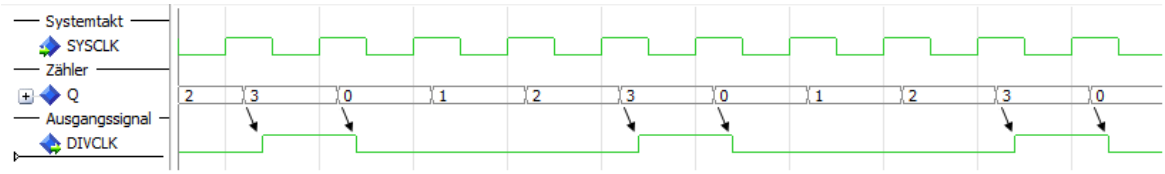


Abbildung 6.2.: Zeitverlauf von DIVCLK im Taktteiler-Modul

Die Abbildung 6.2 zeigt den Systemtakt, den Zähler und das Hilfssignal DIVCLK. Zu sehen ist, dass der Zähler bei jedem SYSCLK um eins erhöht wird. Wenn der Zähler den Zahlenwert 3 erreicht, wird der Pegel des DIVCLK von Low nach High geändert. Mit der nächsten steigenden Flanke von SYSCLK wird der Zähler zurück auf 0 gesetzt und der Pegel von DIVCLK auf Low geändert. Um die Gatterlaufzeit zu simulieren, wurde für jede kombinatorische Logik eine Verzögerung eingefügt.

6.2. Timed-State-Machine-Modul

Das Timed-State-Machine-Modul (TSM) generiert Takte für die Ansteuerung des Audiocodecs sowie die weiteren Signale zur Ansteuerung der Module im Datenpfad.

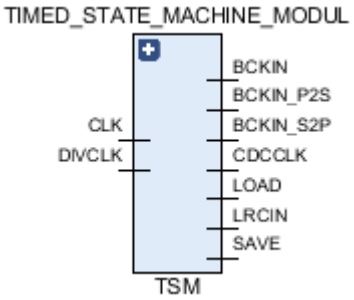


Abbildung 6.3.: Blockschaltbild des TSM Modul

Mit dem zuvor erzeugtem Signal DIVCLK werden die Steuertakte für den Audiocodec erzeugt. Dieser Takt hat eine Frequenz von 20MHz. Um daraus den Takt LRCIN (39062.5 Hz) erzeugen zu können, müssen 512 Takte gezählt werden, wobei 256 Takte lang das Signal High und ebenso lange Low ist, damit ein Tastgrad eingestellt wird. Dasselbe gilt auch für BCKIN mit einer Frequenz von 1.25 MHz und CDCCLK mit einer Frequenz von 10 MHz. Daraus folgernd werden für die Realisierung von LRCIN, BCKIN und CDCCLK ein mod-512-, ein mod-16- und ein mod-2-Zähler benötigt. Da diese Werte ein Vielfaches von 2^x sind, lassen sie sich durch einen einzelnen mod-512-Zähler realisieren. Dazu wird der Zähler bitweise ausgewertet. Der erste Bit steht für den CDCCLK, der vierte für BCKIN und der neunte für den LRCIN-Takt. Dadurch lassen sich Takte generieren, die einen Tastgrad von 50% haben und synchron gesetzt werden.

In der Abbildung 6.4 sind die generierten Takte zu sehen. Durch die großen Frequenzunterschiede der Signale sind die hochfrequenten Signale nicht mehr zu erkennen. In der nachfolgenden Abbildung sind diese vergrößert dargestellt. Insgesamt sind die Taktverhältnisse deutlich erkennbar. In einer Taktperiode von LRCIN befinden sich 32 BCKIN-Taktperioden und in einer BCKIN-Taktperiode 8 CDCCLK-Perioden. Diese geben die Taktverhältnisse wieder.

Es werden vier Steuersignale generiert. BCKIN_S2P und BCKIN_P2S werden für das Schieben der Schieberegister sowohl für den Serien-Parallel- als auch für den Parallel-Serien-Umsetzer verwendet. Mit dem SAVE-Signal wird der Wert im S2P in einem Zwischenregister gespeichert. Mit dem LOAD-Signal werden die Eingangswerte für den P2S eingelesen. Diese Signale lassen sich kombinatorisch aus den Takten BCKIN und LRCIN erstellen.

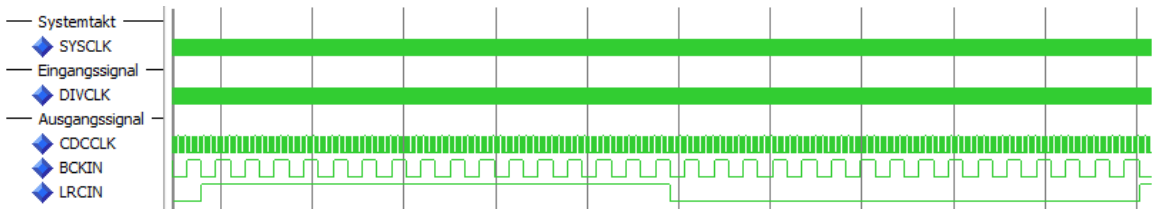


Abbildung 6.4.: Simulation der Steuertakte LRCIN, BCKIN und CDCCLK

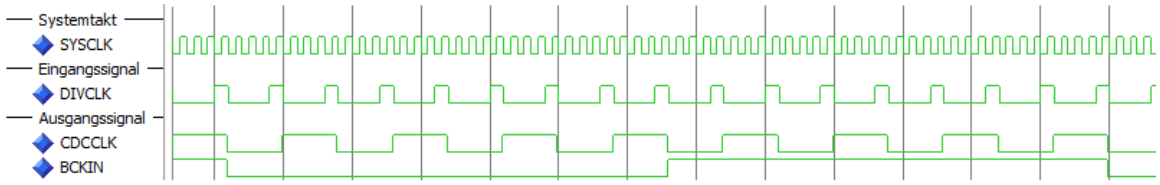


Abbildung 6.5.: Simulation der Steuertakte BCKIN und CDCCLK, vergrößert

Der PCM3006 schreibt bei einer fallenden BCKIN-Flanke einen neuen Wert. Bei einer steigender Flanke kann der Wert sicher vom FPGA gelesen werden. Das BCKIN_S2P-Signal besteht im Grunde aus dem BCKIN Signal. Dazu werden mittels der Impulsverkürzung die steigenden Flanken des BCKIN-Taktes ermittelt. Das daraus entstehende BCKIN_S2P-Signal wird an das S2P-Modul übergeben. Bei jedem Impuls wird das Schieberegister geschoben.

Der PCM3006 liest bei jeder steigender BCKIN-Flanke einen neuen Wert vom seriellen Datenstrom des FPGA. Daher muss schon bei einem fallenden BCKIN Takt der neue Wert geschrieben werden. Dazu werden mittels der Impulsverkürzung die fallenden BCKIN-Flanken detektiert und als BCKIN_P2S ausgegeben. Dieser wird an den P2S gegeben. Bei jedem Impuls wird dieses Schieberegister geschoben.

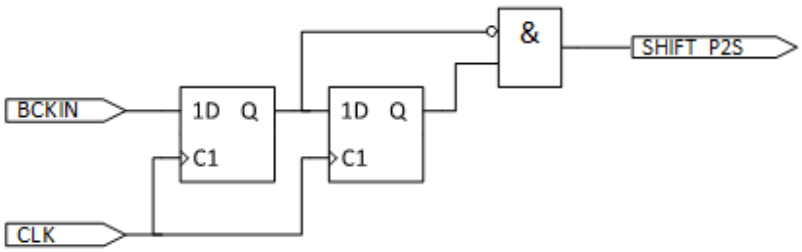


Abbildung 6.6.: Digitale Schaltung zur Impulsverkürzung, beispielhaft für BCKIN_P2S-Signal

Die Abbildung 6.6 zeigt beispielsweise das Erzeugen des BCKIN_S2P-Signals. Für das Erzeugen des BCKIN_P2S-Signals werden die Eingänge des Und-Gatters vertauscht.

Nach 16 Takten der BCKIN sollte das 16-Bit-Datenwort vollständig eingelesen worden sein.

Nach den 16 Takten soll mit dem SAVE-Signal das Schieberegister in ein Zwischenregister übergeben und ausgegeben werden. Das SAVE-Signal wird aus dem LRCIN-Signal generiert. Bei einer Flanke kann man davon ausgehen, dass das Datenwort vollständig eingelesen wurde. Daher werden mittels Impulsverkürzung die Flanken von LRCIN detektiert und als SAVE ausgegeben. Dieses Signal wird an das S2P übergeben.

Mit den Flanken von LRCIN sollen die auszugebenden Signale in den P2S gelesen werden. Dazu werden mit Hilfe der Impulsverkürzung die Flanken des LRCIN detektiert und als LOAD-Signal ausgegeben. Anschließend kann das Signal an den P2S übergeben werden.

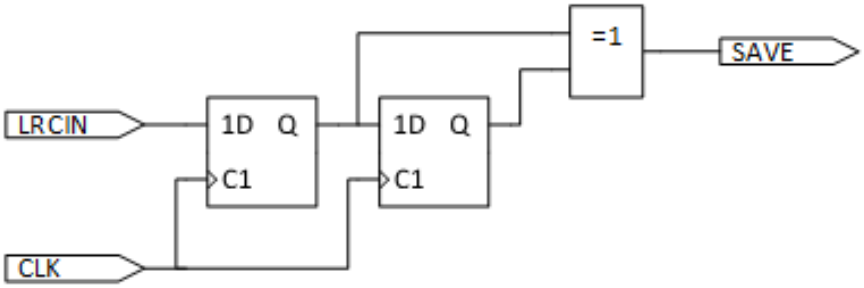


Abbildung 6.7.: Digitale Schaltung zur Impulsverkürzung, beispielhaft für SAVE-Signal

In der Abbildung 6.7 ist die Schaltung zur Erzeugung des SAVE Signals zu sehen. Wie bei der Abbildung 6.6 ist eine Schaltung zur Impulsverkürzung zu sehen, nur mit dem Unterschied, dass statt einem Und-Gatter ein XOR-Gatter für die Detektierung beider Flanken verwendet wurde.

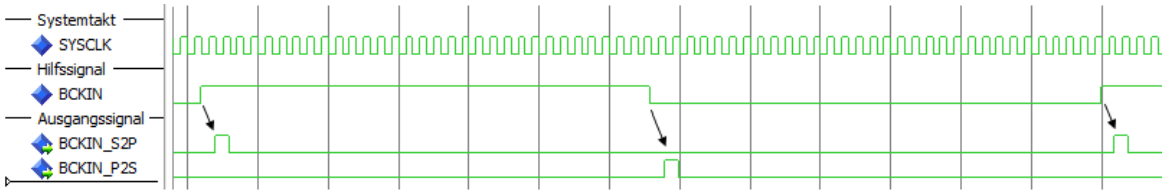


Abbildung 6.8.: Zeitverhalten der BCKIN_S2P und BCKIN_P2S in Abhängigkeit zu BCKIN

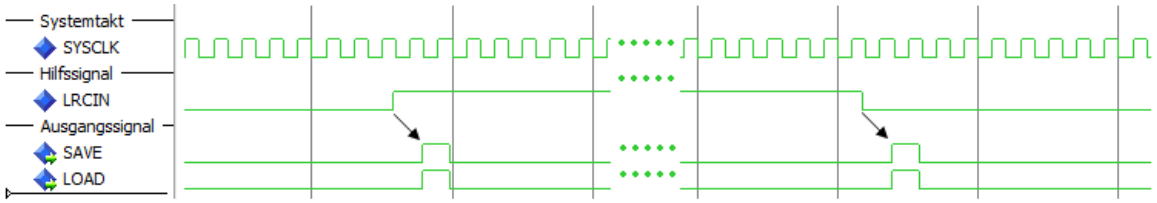


Abbildung 6.9.: Zeitverhalten der SAVE und LOAD in Abhängigkeit zu LRCIN

In der Abbildung 6.8 ist eine Simulation des Zeitverhaltens der Steuersignale BCKIN_S2P und BCKIN_P2S in Abhängigkeit zu BCKIN zu sehen. Ein Pegelwechsel von BCKIN erzeugt

einen Impuls bei BCKIN_S2P oder BCKIN_P2S für die Dauer eines Systemtaktes. Das gleiche Verhalten ist in Abbildung 6.9 mit den Steuersignalen SAVE und LOAD in Abhängigkeit von LRCIN zu sehen. Bei einem Pegelwechsel von LRCIN werden Impulse auf dem LOAD- und dem SAVE- Signal für die Dauer eines SYSCLK generiert.

6.3. Serien-Parallel-Umsetzer-Modul

Wie im vorherigen Kapitel besprochen, wird ein Serien-Parallel-Umsetzer benötigt.

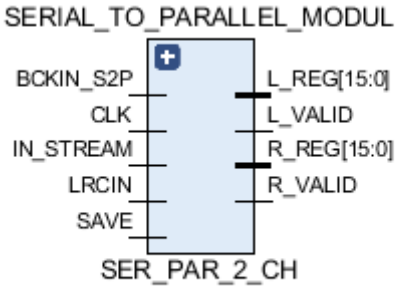


Abbildung 6.10.: Blockschaltbild des Serien-Parallel-Umsetzer-Moduls

In Abbildung 6.10 ist das Blockschaltbild des Serien-Parallel-Umsetzer-Moduls zu sehen. Dieses Modul speichert den 16-Bit seriellen Bitstrom IN_STREAM in ein 16-Bit paralleles Register. Das LRCIN-Signal bestimmt den Kanal, zu dem der kommende Bitstrom gehört. Liegt ein High wird der Linke Kanal übertragen, liegt jedoch ein Low an LRCIN wird der Rechte Kanal übertragen. Mit jedem BCKIN_S2P-Signal wird das IN_STREAM Wert in das Schieberegister geschoben. Wenn das SAVE-Signal anliegt, wird der Schieberegister an das Ausgangsregister abhängig von LRCIN entweder an L-REG oder an R-REG übertragen. Entsprechend wird das dazugehöriger VALID-Signal gesetzt.

In der Abbildung 6.11 ist beispielhaft das Zeitverhalten des Moduls für den linken Kanal zu sehen. Nachdem sich der Pegel von LRCIN und im nächsten Takt der Pegel von SAVE geändert hat, wird das interne Schieberegister L_REG_INT auf das Ausgangsschieberegister L_REG übertragen. Zeitgleich wird das L_VALID für die Zeitdauer eines Systemtaktes auf High gesetzt.

Die Abbildung 6.12 zeigt das Innere des Moduls. Das Modul besteht aus zwei Schieberegistern aus dem Kapitel Grundlagen, jeweils einen für jedem Kanal, zwei Und-Gattern und einem Nicht-Gatter.

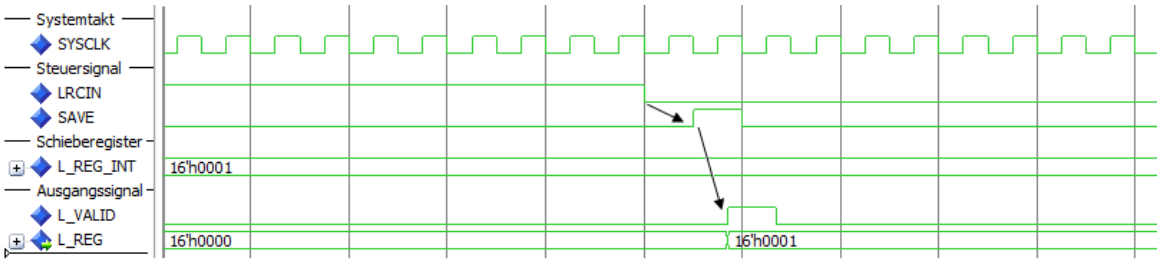


Abbildung 6.11.: Zeitverhalten der Ausgangssignale zum LRCIN-Signal und SAVE-Signal im Parallel-Serien-Umsetzer-Modul

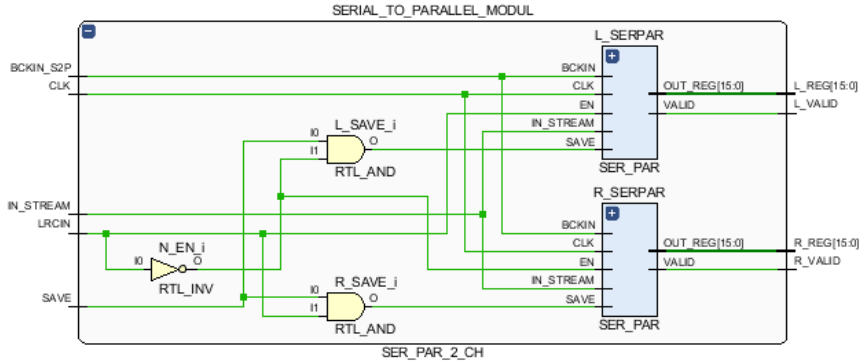


Abbildung 6.12.: Internes Schaltbild des Serien-Parallel-Umwandler-Moduls

Das IN_STREAM ist der serielle Bitdatenstrom. Die Taktdomain ist SYSCLK. Das LRCIN selektiert den Kanal. BCKIN_S2P gibt den Takt zum Schieben der Register. Wenn das SAVE-Signal gesetzt wird, wird das Schieberegister in das Ausgangsregister übergeben und ausgegeben. Zeitgleich wird das VALID-Signal ausgegeben.

6.4. Parallel-Serien-Umsetzer-Modul

Wie oben besprochen, wird hier ein Parallel-Serien-Umsetzer benötigt.

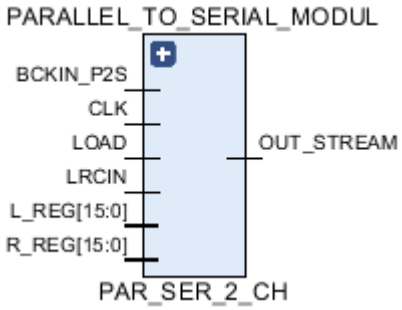


Abbildung 6.13.: Blockschaltbild des Parallel-Serien-Umsetzer-Moduls

In der Abbildung 6.13 ist das Blockschaltbild des Parallel-Serien-Umsetzer-Moduls zu sehen. Wenn an dem LOAD-Signal ein High liegt, wird eines der Register L_REG oder R_REG eingelesen, je nachdem was an LRCIN anliegt. Mit dem BCKIN_S2P Signal wird das Register an OUT_STREAM ausgegeben.

Dieses Modul besteht aus zwei Parallel-Serien-Umwandlern und einem Multiplexer. Das BCKIN_P2S-Signal gibt den Impuls zum Schieben der Schieberegister an. Über den Multiplexer wird der aktuelle Kanal ausgewählt. Die Entscheidung erfolgt über den Wort-Takt LRCIN. Ist der LRCIN Takt High wird das Register für den linken Kanal übergeben, andernfalls wird der rechte Kanal übergeben. Dieser wird bei jedem LOAD-Impuls in das Schieberegister eingelesen. Ausgeben wird das Schieberegister vom MSB zum LSB. Die Schieberegister werden mit Nullen aufgefüllt.

In der Abbildung 6.15 ist die Simulation des Zeitverhaltens des Parallel-Serien-Umsetzers für den linken Kanal zu sehen. Mit der Pegeländerung am LRCIN wird im nächsten Takt das SAVE-Signal geändert und im darauffolgenden Takt der linke Kanal in das interne Register eingelesen.

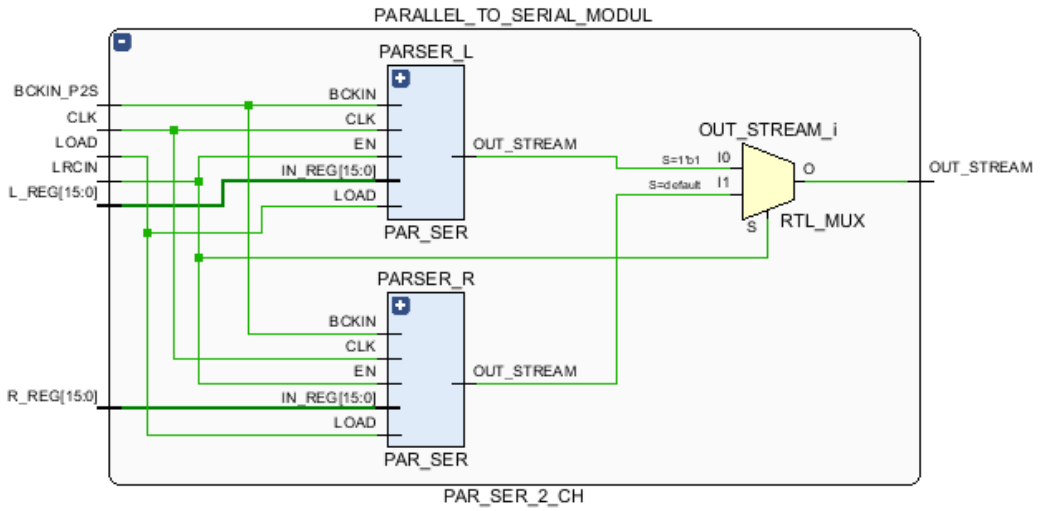


Abbildung 6.14.: Internes Schaltbild des Parallel-Serien-Umsetzer-Modul

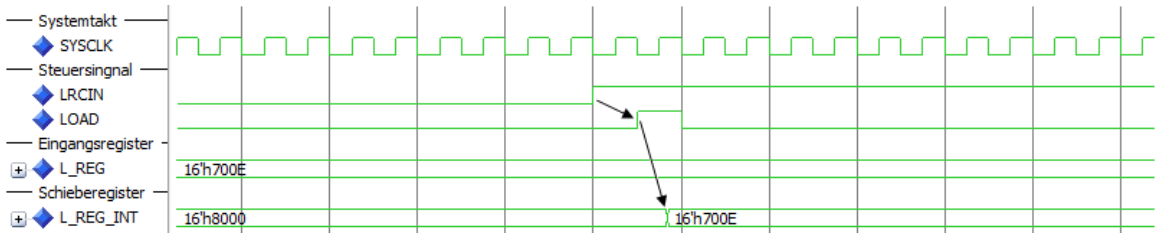


Abbildung 6.15.: Zeit- und Signalverhalten der Eingangs- und Ausgangssignalen im Parallel-Serien-Umsetzer-Moduls

6.5. Evaluation des Basiscodes

In dieser Sektion wird der VHDL-Basiscode evaluiert. Dazu wird zuerst mit Hilfe von Modelsim das Zeitverhalten überprüft. Als nächstes wird mit Vivado synthetisiert und implementiert. Die Zeitreport und Simulationreport wird analysiert. Zum Schluss wird das Programm auf den FPGA-Chip hochgeladen und mit Signalgenerator, Oszilloskop und Spektrum-Analysator getestet.

6.5.1. Zeitverhalten mit ModelSim

Für den Gesamttest werden die einzelnen Module zusammengetragen. Die VHDL-Struktur wird wie in Abbildung 4.4 geschrieben und eine dazu passendes Testbench geschrieben. Es ist das Zusammenspiel der Signale zu analysieren. Hierbei wird im Signalverarbeitungsmodul das Eingangsregister mit dem Ausgangsregister kurzgeschlossen. Zuerst wird das Gesamtverhalten des Basiscodes betrachtet.

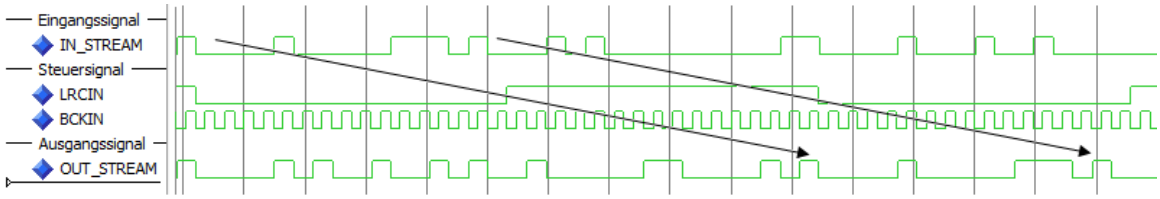


Abbildung 6.16.: Signalübertragung des linken Kanals im FPGA-Chip, wenn im Signalverarbeitungsmodul die Eingangsregister mit den Ausgangsregister kurzgeschlossen sind

In der Abbildung 6.16 ist die Signalübertragung des linken Kanals zu sehen. Der eingelesene Datenstrom von 16-Bit wird während der nächsten linken Phase in der gleichen Form ausgegeben.

Als nächstes wird die zeitliche Abhängigkeit von BCKIN und dem Ausgangssignal OUT_STREAM untersucht. Laut des Timing-Diagramms (Abbildung A.1) aus dem Datenblatt [1] darf die Zeitverschiebung nicht mehr als 40 ns betragen.

In der Abbildung 6.17 ist die zeitliche Abhängigkeit des BCKIN von OUT_STREAM zu sehen. Bei dieser Simulation mit 100 MHz beträgt die Zeitverschiebung etwa 22 ns. Das ist genau die Zeitdauer von zwei Systemtakten und der Gatterlaufzeit von 2 ns. Analog dazu würde bei einem Systemtakt von 80 MHz die Zeitdauer 27 ns dauern, mit der Annahme, dass die kombinatorische Logik 2 ns dauert. Diese Zeitdauer ist weit unter dem angegebenen Wert und somit geeignet für die Anwendung.

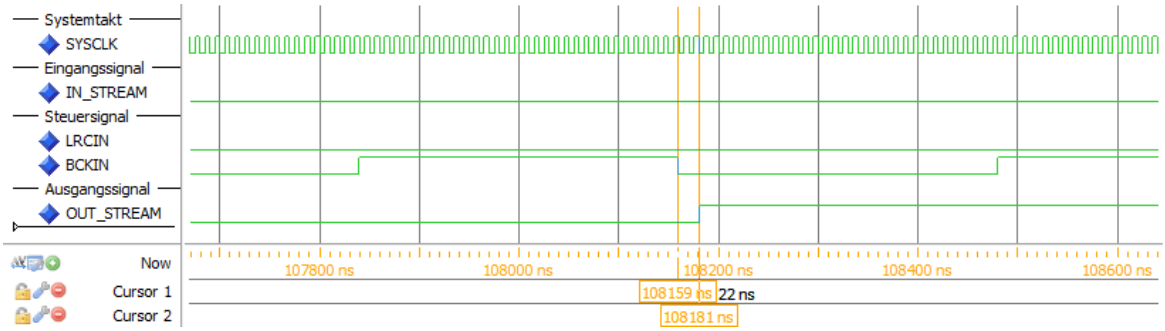


Abbildung 6.17.: Zeitliche Verschiebung vom BCKIN-Signal zum OUT_STREAM

6.5.2. Simulation mit Vivado

Als nächstes wird die Entwicklungsumgebung Vivado von Xilinx verwendet. Vivado ist eine Software, die VHDL synthetisieren und analysieren kann. Für die Arbeit wurde die Version v2017.2 verwendet. Ein neues Projekt wird in Vivado erstellt. Das FPGA-Chip XC7A100T-CSG324 wird aus der Datenbank ausgewählt, und alle VHDL-Codes werden hinzugefügt. Als erstes kann eine funktionale Simulation des VHDL-Codes in Vivado vorgenommen werden. Jedoch ist die Simulation nicht mehr nötig, da die Funktionalität mit ModelSim überprüft wurde. Vor der eigentlichen Synthese wird der VHDL-Code auf Schaltplanebene untersucht. Dazu wird in Vivado RTL Analysis (Register-Transfer Level) ausgeführt. Diese wird mit der Erwartung (vgl. Bild 4.4) verglichen.

Im Anhang B.2 ist die RTL-Sicht der Basisstruktur zu sehen. Wie erwartet, sind die fünf beschriebenen Module sowie alle Signale zu sehen. Auf der linken Seite sind die Eingangs- und auf der rechten Seite die Ausgangssignale zu sehen.

Als nächstes wird in Vivado Run-Synthesis durchgeführt. Zuvor wird eine Datei mit der Dateierweiterung *.xdc in das Projekt eingebunden. In dieser Datei sind die Pinbelegung und die Einstellung einzelner Pins sowie der Taktpin und die Takteinstellung hinterlegt. Danach kann die VHDL-Synthese durchgeführt werden.

Nach der erfolgreichen Synthese wird in Vivado Run-Implementation ausgeführt. In diesem Schritt erfolgen die Platzierung und Verdrahtung der FPGA-Ressourcen. Zudem wird das Design optimiert [2].

In der Zusammenfassung der Implementation kann man sich die Rechenzeiten anschauen. Unter dem Teilfenster Design Timing Summary kann man die Taktreserve (Worst Negative Slack) entnehmen. Für diese Simulation wurde eine Zeit von 10.279 ns ermittelt. Bei einer Taktperiodendauer von 12.5 ns beträgt der kritische Pfad 2.221 ns. Somit könnte die Schaltung ohne Probleme bis 450 MHz betrieben werden, unter der Bedingung, dass zuvor das

Taktteiler Modul angepasst wird. Unter Utilization sind die benötigten Ressourcen aufzufinden. Es wurden 42 LUTs und 111 Flipflops benutzt (vrgl. 6.18).

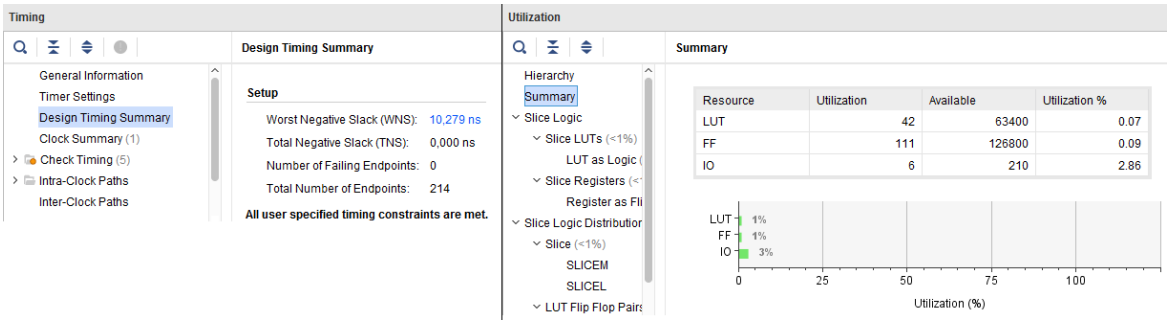


Abbildung 6.18.: Timing-Analyse und benötigte Ressourcen für den Grundprogramm für die Steuerung der Audiocodec-Platine

Zum Schluss muss das Programm auf den FPGA-Chip hochgeladen werden. Dazu wird Generate Bitstream ausgeführt. Vivado erstellt aus der Implementation die Programmierdatei mit der Dateierweiterung *.bit. Mit Open Target wird der FPGA-Chip ausgewählt und mit Program Device die Programmierdatei hochgeladen. Abschließend kann die Hardware getestet werden.

6.5.3. Evaluation der VHDL-Basisstruktur

Als erstes werden die Taktsignale CDCCLK, LRCIN und BCKIN untersucht. Dazu werden sie mit Tastköpfen auf der Audio-Codec-Board Platine an den dafür vorgesehenen 2mm Steckbuchsen angegriffen und auf einem Oszilloskop angezeigt.

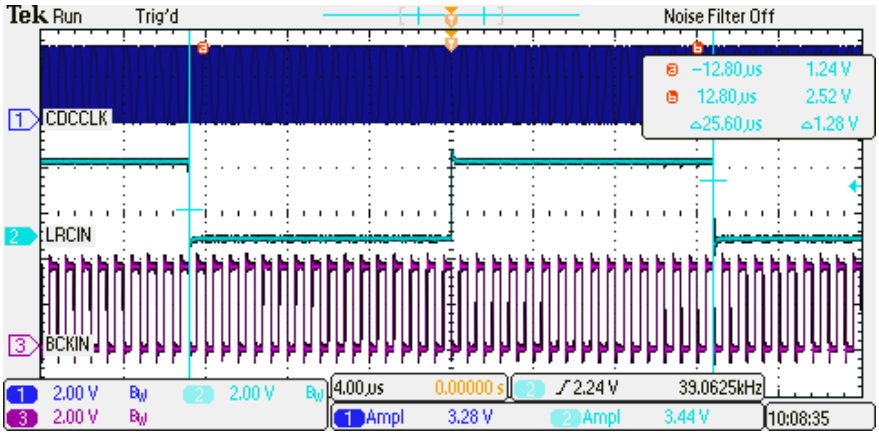


Abbildung 6.19.: Taktsignale CDCCLK, LRCIN und BCKIN an der Audiocodec-Platine

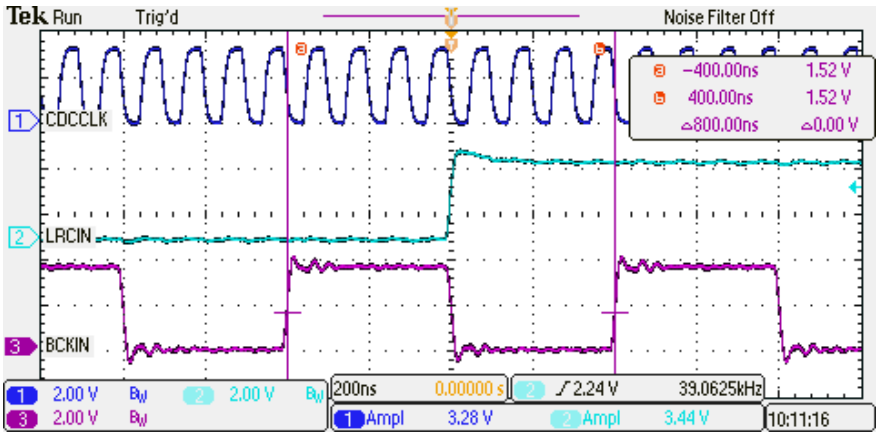


Abbildung 6.20.: Taktsignale CDCCLK, LRCIN und BCKIN an der Audiocodec-Platine, vergrößert

In der Abbildung 6.19 und in der vergrößerten Ansicht in Abbildung 6.20 sind die Taktsignale zu sehen. Sie haben die gewünschten Taktlängen.

Die Gesamtverstärkung wird mit einem Spektrum-Analysator untersucht. Ein Spektrum-Analysator ist ein Messgerät, mit dem das Frequenzspektrum eines Systems dargestellt werden kann. Dazu erzeugt der Spektrum-Analysator ein Sinussignal. Das Signal wird an den Eingang des zu testenden Systems angelegt. Der Ausgang wird dann zum Eingang des Spektrum-Analysators geführt. Die Frequenz des Sinussignals wird Schritt für Schritt von einer niedrigen zu einer hohen Frequenz hochgestellt. Dabei wird das Ausgangssignal erfasst und aufgezeichnet.

Der Spektrum-Analysator wird an die 2mm-Steckbuchsen am Eingang der Platine angeschlossen. Die 2mm Steckbuchsen am Ausgang werden zur Spektrum-Analysators geführt. Die Grenzen wurden bei 10Hz und 32Hz gewählt.

Bild 6.21 zeigt den Frequenzgang der Platine. Zu sehen ist ein Tiefpassverhalten. Zu sehen ist, dass der Frequenzgang bis etwa 18 kHz annähernd konstant ist. Ab 18 kHz wird das Signal allmählich auf -50 dB gedämpft. Das sind in etwa die Audiocodec interne Grenzen. Diese liegen bei 0.454 fs für den Durchlassbereich und 0.583 fs für den Sperrbereich. Bei fs = 39062.5 Hz ergibt das 17.7 kHz und 22.8 kHz.

Als nächstes wird die Bearbeitungslaufzeit des PCM3006 gemessen. Dazu wird ein Rechteckimpuls an den Eingang der Platine angelegt und am Ausgang mit einem Oszillator gemessen. Die Zeitdifferenz zwischen Eingangs- und Ausgangssignal ist die Laufzeit durch den Audiocodec. Im Datenblatt [1] ist verzeichnet, dass der digitale Filter am ADU eine Verzögerungszeit von 17.4/fs hat. Der digitale Filter am DAU hat eine Verzögerungszeit von 11.1/fs. Mit der Bearbeitungszeit im Signalverarbeitungsmodul von fs hat das gesamte System eine

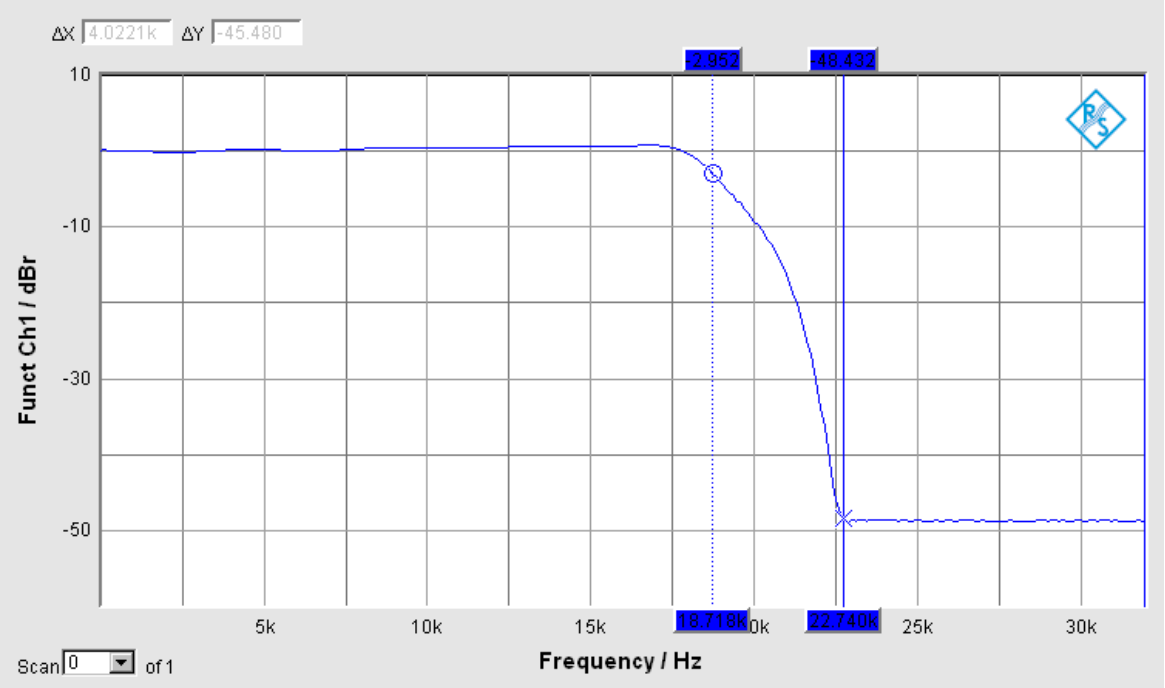


Abbildung 6.21.: Frequenzgang der Audiocodec-Platine von 10 Hz bis 48kHz

Verzögerungszeit von 29.5/fs. Bei eine Abtastrate von 39062.5Hz bedeutet das, dass die erwartete Verzögerungszeit etwa 755.2 us beträgt.

In der Abbildung 6.22 ist eine steigende Flanke des Eingangssignals und das Systemantwort der Platine am Ausgang zu sehen. Die Zeit zwischen den beiden Flanken ist die Verzögerungszeit des gesamten Systems. Die gemessene Zeit beträgt etwa 766 us. Das sind etwa 29.9/fs und stimmt ungefähr mit dem erwarteten Wert von 29.5/fs überein. Für das menschliche Gehör ist die Verzögerung nicht mehr wahrnehmbar. Zuletzt werden die möglichen Spannungsgrenzen der Platine getestet. Die Platine ist für eine Spannung von -1.41 bis 1.41 V ausgelegt. Um dies zu testen, wird ein Sinussignal an den Eingang angelegt, mit einer Amplitude, die höher ist als 1.414 V.

Bei einer Einspeisung eines Sinussignals mit einer Amplitude von 1.41 V wird das Signal noch originalgetreu ausgegeben. Bei einem Sinussignal mit einer Amplitude von 1.45 V ist eine Abflachung der Spitzen zu erkennen.

Das VHDL-Modell kann noch Ressourcensparend optimiert werden. Für diese Arbeit war die Optimierung des VHDL-Modell nicht gefordert. Dieses Modell dient zum Testen der Platine.

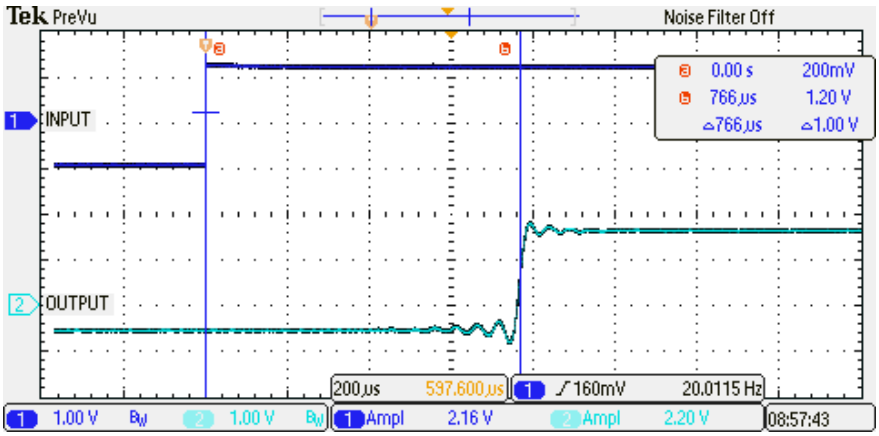


Abbildung 6.22.: Reaktion der Audiocodec-Platine auf ein Impuls

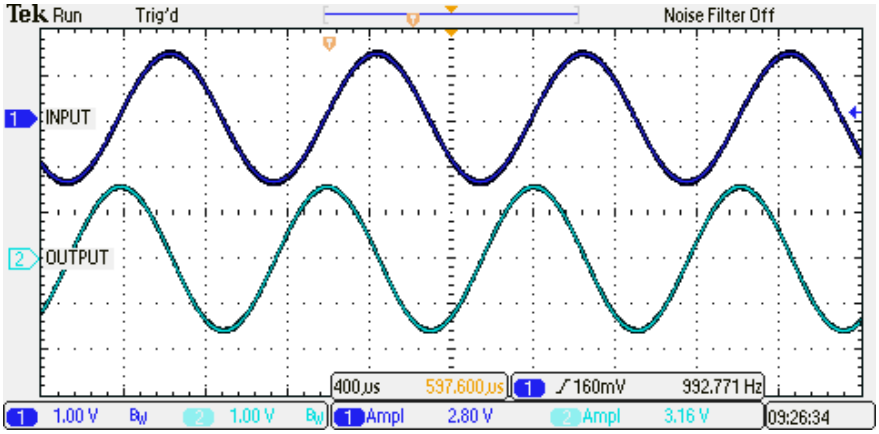


Abbildung 6.23.: Ausgangssignal bei einer Einspeisung von einem Sinussignal mit einer Amplitude von 1.41V

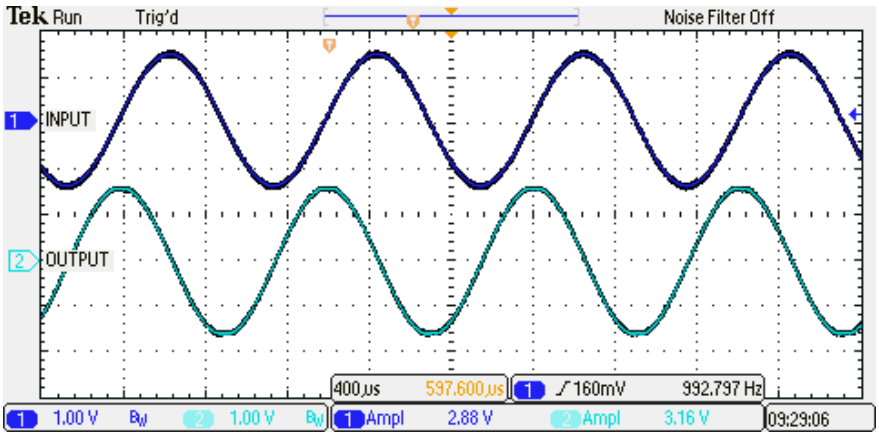


Abbildung 6.24.: Ausgangssignal bei einer Einspeisung von einem Sinussignal mit einer Amplitude von 1.45V

7. Signalverarbietung mit High-Level Synthese

In diesem Kapitel werden die ersten Schritte mit der Vivado HLS Sowftware beschrieben und entsprechend durchgeführt. Mit dieser Software bietet Xilinx die Möglichkeit aus einem C-Code einen VHDL-Code zu synthetisieren. Um die Komplexität und die Vereinfachung durch Vivado HLS zu zeigen wird beispielhaft ein FIR-Filter zuerst mit VHDL geschrieben und getestet und anschließend mit einem C-Code über Vivado HLS synthetisiert. Der FIR-Filter mit VHDL wird nach dem Buch VHDL-Synthese [12] designet, synthetisiert und implementiert. Alsdann wird mit Hilfe von HLS ein FIR-Filter in C entwickelt und dieser mit dem selbst geschriebenen FIR Filter verglichen und ausgewertet. Im Anschluss wird mit Hilfe von Vivado HLS der Audioeffekt Vibrato geschrieben, entwickelt, implementiert und ausgewertet.

7.1. FIR-Filter in VHDL

Mit Hilfe von MATLAB wird der Tiefpass-FIR-Filter entworfen. Die Funktion `fir1` berechnet die Filterkoeffizienten anhand der Filterordnung N und der normierten Eckfrequenzen W. Mit Hilfe der Funktion `kaiserord` wird N abgeschätzt und liefert weitere Parameter für die `fir1`-Funktion. Die Funktion `kaiserord` benötigt die Eckfrequenzen, die maximalen Ripple für den Durchlass- und Sperrbereich sowie die Abtastrate. Die Abtastrate beträgt bei einem Systemtakt bei 80 MHz etwa 39.1 kHz. Die Eckfrequenzen wurden bei $1/10fs$ und $2/10fs$ gewählt. Die Filterordnung wurde mit MATLAB auf 23 geschätzt und hat somit 24 Koeffizienten. Die von MATLAB errechneten Koeffizienten sind Dezimalzahlen. Diese werden zuerst auf 2^{15} normiert.

In der Abbildung 7.1 ist der Frequenzgang des FIR-Filters zu sehen. Die Frequenzachse ist auf fs/2 normiert. Zu sehen ist ein Tiefpass verhalten, mit den Grenzen bei 0.1 fs und 0.2 fs. Ab 0.2 fs hat der Filter eine Dämpfung von 40 dB.

Als nächstes wird ein VHDL-Code für einen FIR-Filter geschrieben. Der FIR-Filter wird in Direktform entworfen. Alternativ kann die sogennante Linear-Phasen-Struktur entworfen werden, jedoch gibt es fast keinen Unterschied bei der Bearbeitungszeit. Mit der Linear-Phasen-Struktur lässt sich etwa die Hälfte der benötigten Anzahl an LUT einsparen[12].

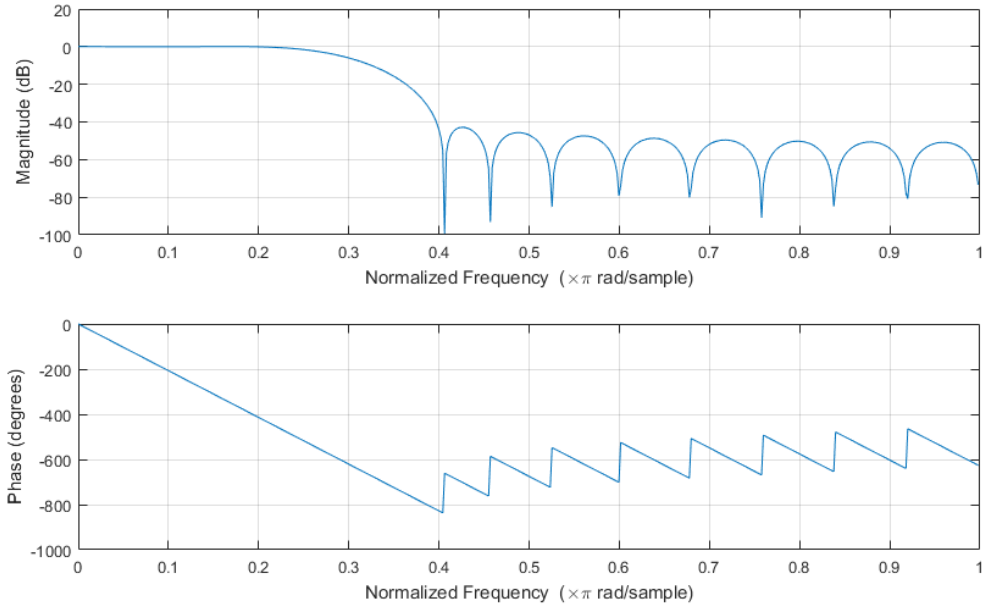


Abbildung 7.1.: Frequenzgang des FIR-Filters

Es wird ein Filter mit einer balancierten Addiererbaum-Struktur mit 24 Koeffizienten entworfen. Durch die for-loop-Schleife in VHDL lassen sich die Addierstufen einfach realisieren. Ein Beispiel für einen VHDL-Code für einen FIR-Filter wurde aus dem Buch VHDL-Synthese [12] entnommen und angepasst.

Der VHDL-Code für den FIR-Filter soll auf den FPGA-Chip hochgeladen werden. Dazu wird Vivado benutzt. Zunächst wird ein neues Projekt erstellt und der VHDL-Code für die Basisstruktur geladen. Anschließend wird der FIR-Filter Code in das Projekt eingefügt. Mittels `port map` wird der Filter in das Signalverarbeitungs-Modul eingefügt. Anschließend wird das Projekt synthetisiert und implementiert.

Unter Report Utilization kann ein Einblick in die benötigten Ressourcen gewonnen werden. Dieses Signalverarbeitungsmodul benötigt für zwei FIR-Filter zusätzlich 750 LUTs und 40 Flipflops.

Anschließend wird mit Vivado die Programmierdatei erstellt und auf den FPGA-Chip hochgeladen. Des Weiteren wird mit einem Spektrum-Analysator der Amplitudengang gemessen. Zu erwarten ist ein Tiefpassverhalten. Der Durchlassbereich soll bis 1/10 fs sein. Ab 2/10 fs sollen die Frequenzen sperren (40dB).

In der Abbildung 7.2 ist der Frequenzgang des selbstgeschriebenen FIR-Filters zu sehen. Leider sieht er nicht so aus wie in Abbildung 7.1 erwartet. Durch mangelnde Zeit war es

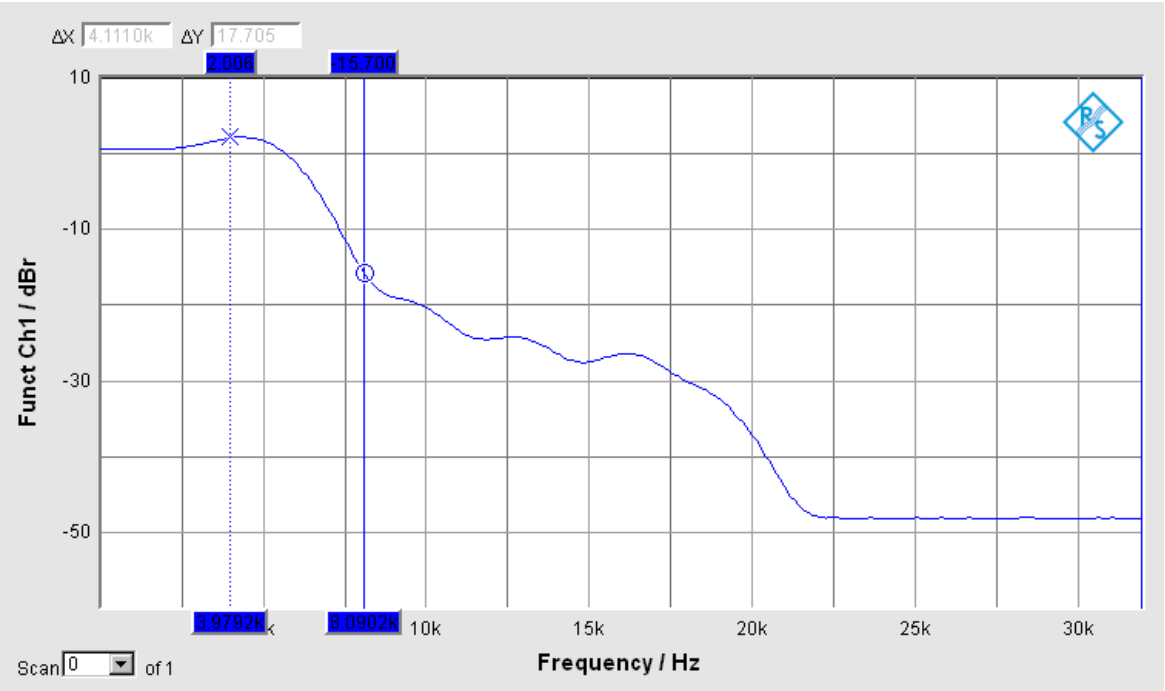


Abbildung 7.2.: Frequenzgang des FIR-Filters

nicht mehr möglich diesen Fehler zu analysieren und nachzuverfolgen, da dieser Fehler spät erkannt wurde.

7.2. FIR-Filter mit HLS

Die Software Vivado HLS generiert aus einem C/C++ einen VHDL-Code. Der C/C++ Code besteht aus zwei Funktionen, aus der `main` und einer weiteren Funktion. Die zweite Funktion stellt den Code dar, der in VHDL synthetisiert werden soll. Die Übergabeparameter werden als Ein- und Ausgänge der Entity-Schnittstelle übersetzt. Call-by-Value-Parameter werden als Eingänge definiert und Call-by-Reference-Parameter werden je nach Anwendung als Eingang oder als Ausgang definiert. Der Rückgabewert der Funktion wird in der Entity als Ausgang umgesetzt. Die Entity übernimmt den Namen der C-Funktion. Die `main`-Funktion wird unter anderem als Testbench verwendet. In ihr werden die Signale für die Entity-Schnittstelle generiert und übergeben. Die Antwort der Funktion kann mit erwarteten Werten verglichen und somit getestet werden.

Unter den Beispielen von Vivado HLS befindet sich unter anderem ein `fir`-Code in C für den Entwurf eines FIR-Filters. Mit Hilfe dessen soll ein FIR-Filter im VHDL-Code generiert

werden. Die Filtereigenschaften werden aus der vorherigen Sektion übernommen, sodass die Filterordnung und die Koeffizienten gleich bleiben. Zusätzlich werden mit MATLAB ein Signal zum Testen des FIR-Filters erstellt und die erwarteten Ausgangswerte berechnet. Diese Werte werden in einer separaten Datei abgespeichert und in die `main`-Funktion eingelesen. Die Werte werden der `fir`-Funktion übergeben und mit dem erwarteten Wert verglichen.

Listing 7.1: `fir`-Funktion in C

```
1  void fir (
    data_t *y,
3   data_t x
   ) {
5   static data_t shift_reg[N];
    acc_t acc;
7   data_t data;
    int i;
9   acc=0;
    Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
11   if (i==0) {
      shift_reg[0]=x;
13    data = x;
     } else {
15    shift_reg[i]=shift_reg[i-1];
      data = shift_reg[i];
17   }
     acc+=data*h[i];
19  }
    *y=(acc >> 15);
21 }
```

Nachdem der C-Code erfolgreich getestet wurde, kann er synthetisiert werden. Zuvor werden mittels Direktiven im C-Code Optimierungen und Anpassungen beim Synthetisieren des VHDL-Codes vorgenommen. So lassen sich beispielsweise die Entity-Schnittstelle anpassen oder die Abarbeitung einer for-Schleife einstellen.

Unter INTERFACE in Directive lässt sich das I/O Schnittstelle optimieren. Dadurch hat man die Möglichkeit, bei ap_none keine zusätzlichen Signal zu erzeugen oder bei ap_hs Handshake-Signale zu erzeugen, bestehend aus einem Start-Signal (ap_start), einem Fertig-Signal (ap_done), einem Beschäftigt-Signal (ap_idle) und einem Bereit-Signal (ap_ready). Mit dem Start-Signal startet das Modul die Berechnung. Die anderen drei Signale geben an, fertig berechnet hat, noch beschäftigt ist oder ob das Modul bereit ist, eine neue Berechnung durchzuführen[20][21].

Die for-Schleife kann optimiert werden, indem die Direktiven eingestellt werden, wie die

Schleife umgesetzt werden soll. Es bestehen dabei die Möglichkeiten der Pipelining-Schleife und der Unrolling-Schleife. Bei Ersterer wird die Schleife sequentiell abgearbeitet. Die nächste Schleifeniteration kann erst beginnen, wenn die vorherige Berechnung abgeschlossen wurde. Bei der Unrolling-Schleife werden mehrere Kopien der Schleifenrumpf erstellt. Diese wird dann durchlaufen. Die erste Schleife benötigt weniger Ressourcen, aber dafür mehrere Takte, um die Schleife komplett zu bearbeiten. Die zweite Schleife benötigt weniger Takte, jedoch werden mehr Ressourcen benötigt als bei der Pipelining-Schleife[20][21][22].

Die `fir`-Funktion wird nacheinander mit beiden Schleifen synthetisiert.

Performance Estimates

Timing (ns)

Clock		pipeline	unrolled
ap_clk	Target	10.00	10.00
	Estimated	8.53	9.12

Latency (clock cycles)

		pipeline	unrolled
Latency	min	52	3
	max	52	3
Interval	min	53	4
	max	53	4

Utilization Estimates

	pipeline	unrolled
BRAM_18K	0	0
DSP48E	1	24
FF	199	1113
LUT	147	251

Abbildung 7.3.: Die berechneten Ressourcen und Takte für die Einstellungen Pipelining-Schleife und Unrolling-Schleifen

Die Abbildung 7.3 zeigt den Ressourcenverbrauch und die Latenzzeiten der beiden Simulationen. Deutlich wird, dass bei der Pipelining-Schleife weniger Ressourcen benötigt werden. Jedoch werden dafür 53 Takte benötigt. Die Unrolling-Schleife benötigt mehr, Ressourcen jedoch werden nur vier Takte benötigt.

Diese beiden Varianten stellen die beiden Extreme dar. Alternativ können Blöcke eingestellt werden, die aus mehreren Unrolling-Schleifen bestehen und sequenziell abgearbeitet werden.

Der selbst erstellte FIR-Filter benötigt etwa 750 LUT und 40 FF. Im Vergleich mit dem selbst erstellten FIR-Filter ist erkennbar, dass der selbst erstellte FIR-Filter mehr LUT, dafür aber weniger FF benötigt.

Der VHDL-Code des FIR-Filters wird in das Signalverarbeitungsmodul eingebunden. Anschließend wird der gesamte VHDL-Code synthetisiert und auf den FPGA hochgeladen. Mit einem Spektrum-Analysator wird der Frequenzgang des FIR-Filters gemessen.

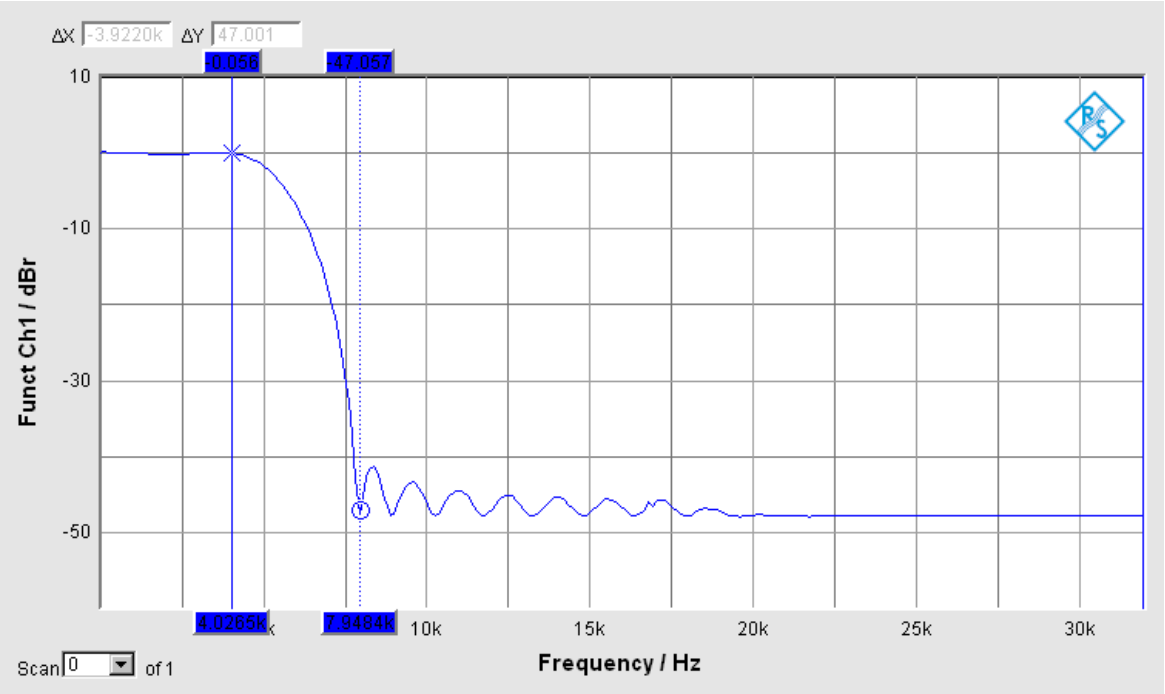


Abbildung 7.4.: Frequenzgang des mit HLS erzeugten FIR-Filters

In der Abbildung 7.4 ist der Frequenzgang des FIR-Filters zu sehen. Der VHDL-Code wurde mit Vivado HLS erzeugt. Der Frequenzgang stimmt mit der Abbildung 7.1 überein. Der Durchlassbereich geht bis etwa 0.1 fs. Ab 0.2 fs gibt es eine Dämpfung von weniger als 40 dB.

7.3. Realisierung des Vibrato-Effektes mit HLS

Für das Testen des Audiocodec-Boards wird ein Audioeffekt verwendet. Zunächst wird MATLAB der Vibrato-Effekt aus den Kapitel Grundlagen getestet. Dazu wird der Code dafür aus dem Buch [4] genommen und erweitert.

Listing 7.2: MATLAB-Code für den Vibrato-Audioeffekt[4]

```
function y=vibrato(x,SAMPLERATE,Modfreq,Width)

Delay=Width;                    % basic delay of input sample in sec
DELAY=round(Delay*SAMPLERATE); % basic delay in # samples
WIDTH=round(Width*SAMPLERATE); % modulation width in # samples

MODFREQ=Modfreq/SAMPLERATE; % modulation frequency in # samples
LEN=length(x);              % # of samples in WAV-file
L=1+WIDTH*2;                % length of the entire delay
Delayline=zeros(L,1);       % memory allocation for delay
y=zeros(size(x));           % memory allocation for output vector

for n=1:(LEN-1)
    M=MODFREQ;
    MOD=sin(M*2*pi*n);
    TAP=1+DELAY+WIDTH*MOD;
    i=floor(TAP);
    frac=TAP-i;
    Delayline=[x(n);Delayline(1:L-1)];
    %---Linear Interpolation-----------------------------
    y(n,1)=Delayline(i+1)*frac+Delayline(i)*(1-frac);
end
```

Zuerst müssen die Oszillation des Zeigers in Hertz (Width) sowie die Zeit, um die sich das Eingangs zu Ausgangssignal verzögern soll, in Sekunden (Modfreq) gewählt werden. Typischer Weise liegt die Oszillation des Zeigers zwischen 5 und 14Hz und die Verzögerungszeit zwischen 5 und 10ms. Zudem muss die Abtastzeit (SAMPLERATE) angegeben werden. Aus den oben gewählten Werten wird die Breite des Speichers für die Verzögerungskette (Delayline) bestimmt und reserviert sowie die Frequenz für die Modulation (MODFREQ) des Zeigers berechnet. Danach wird ein Speicher in der selben Breite wie der Eingangswerte für die Ausgangswerte reserviert. Im Anschluss wird in einer for-Schleife wird zuerst die Position des Zeigers (MOD) sowie den Wert für den Hilfsteiler (frac) bestimmt und dann den Aktuellen Eingangswert in die Delayline geschoben und anschließend durch lineare Interpolation der Ausgangssignalwert bestimmt.

Für die nachfolgenden Tests werden die nachfolgenden Eigenschaften für die Entwicklung des Effektes gewählt.

- Abtastfrequenz: 39062.5 Hz (An SYSCLK von 80 MHz angepasst)
- Oszillation beträgt: 5Hz

- Delayline beträgt: 5 ms
- Interpolationsart: Lineare Interpolation

Um den Vibrato Effekt zu testen wird ein Sinussignal mit einer Frequenz von 1kHz generiert.

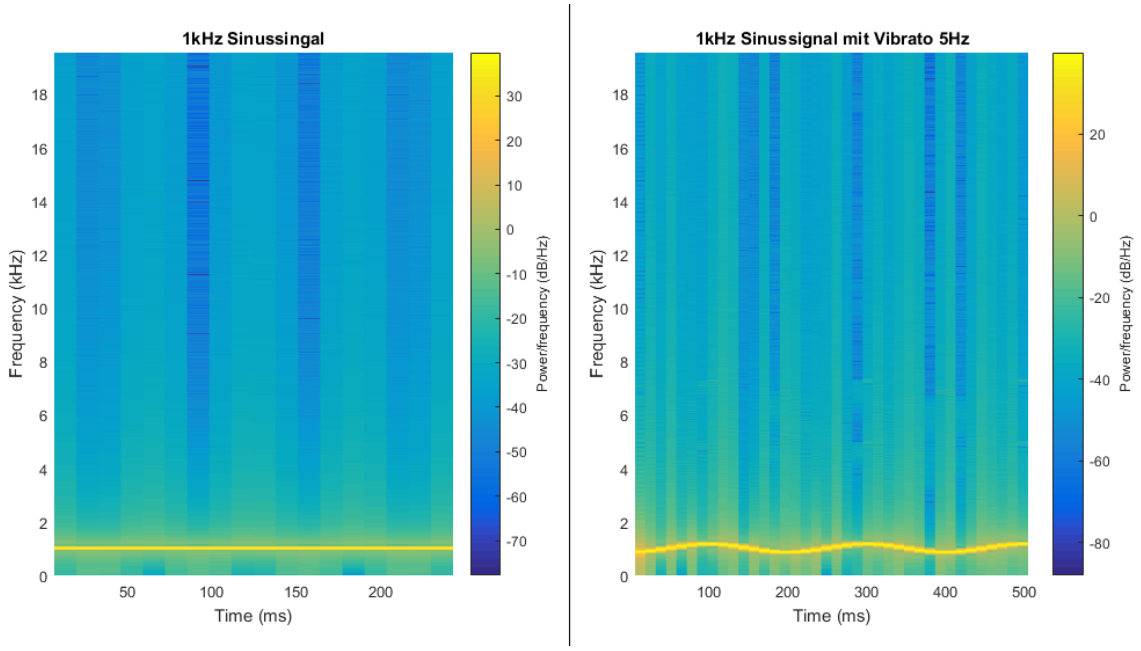


Abbildung 7.5.: Frequenzspektrum eines 1kHz Sinussignals (links) sowie eines mit 5Hz modulierten 1kHz Sinussignals (rechts)

In der Abbildung 7.5 ist links das Frequenzspektrum über die Zeit des Eingangssignals und rechts das Frequenzspektrum über die Zeit mit dem Vibrato-Effekt zu sehen. Das linke Signal ist ein 1kHz Sinussignal und das rechte Signal ein mit 5Hz moduliertes 1kHz Sinussignal. Wie erwartet ist im linken Bild eine konstante Frequenz von 1kHz über die Zeit zu sehen und im rechten Bild im Spektrum ein Signal, der sich um 1kHz schwingt, zu erkennen.

Aus dem MATLAB-Code vom Listing 7.2 soll ein C-Code extrahiert werden. Dazu wird der Inhalt der for-Schleife zu einer C-Funktion umgewandelt. Dabei werden die Parameter des MATLAB-Codes sowie die Länge der Verzögerungskette fest programmiert.

Einige Standart C-Bibliotheken können für die Programmierung verwenden, jedoch empfiehlt sich die HLS eigenen Bibliotheken zu benutzen, weil sie für die VHDL-Synthese mit HLS entwickelt wurden. So existiert für den Datentypen float und double die ap_fixed.h Bibliothek und statt der math.h kann hls_math.h verwendet werden.

Die hls_math.h beinhaltet viele mathematischen Funktionen, jedoch kam es mit der Sinus-Funktion bei der VHDL-Synthese zu Problemen. Aus diesem Grund wurde die Sinusfunktion für eine ganze Periode zu den einzelnen n-Stellen ausgerechnet und in einen Speicher abgelegt. Diese werden dann über eine Header-Datei in das Projekt eingebunden. Die Berechnung für `frac` erfolgt wie in Listing 7.2. Anschließend erfolgt das Scheiben von Delayline und das einlesen des aktuellen Eingangssignalwerts. Zum Schluss wird der Ausgangssignalwert durch den Zeiger und durch lineare Interpolation ermittelt und die Laufvariabel n hochgezählt.

In den `main`-Funktion wird ein kleines Testbench geschrieben und ausgewertet. Als Eingangssignal wird ein Sinussignal generiert und an die `vibrato`-Funktion übergeben. Das Ausgangssignal wird in eine Datei eingespeichert und in MATLAB ausgewertet.

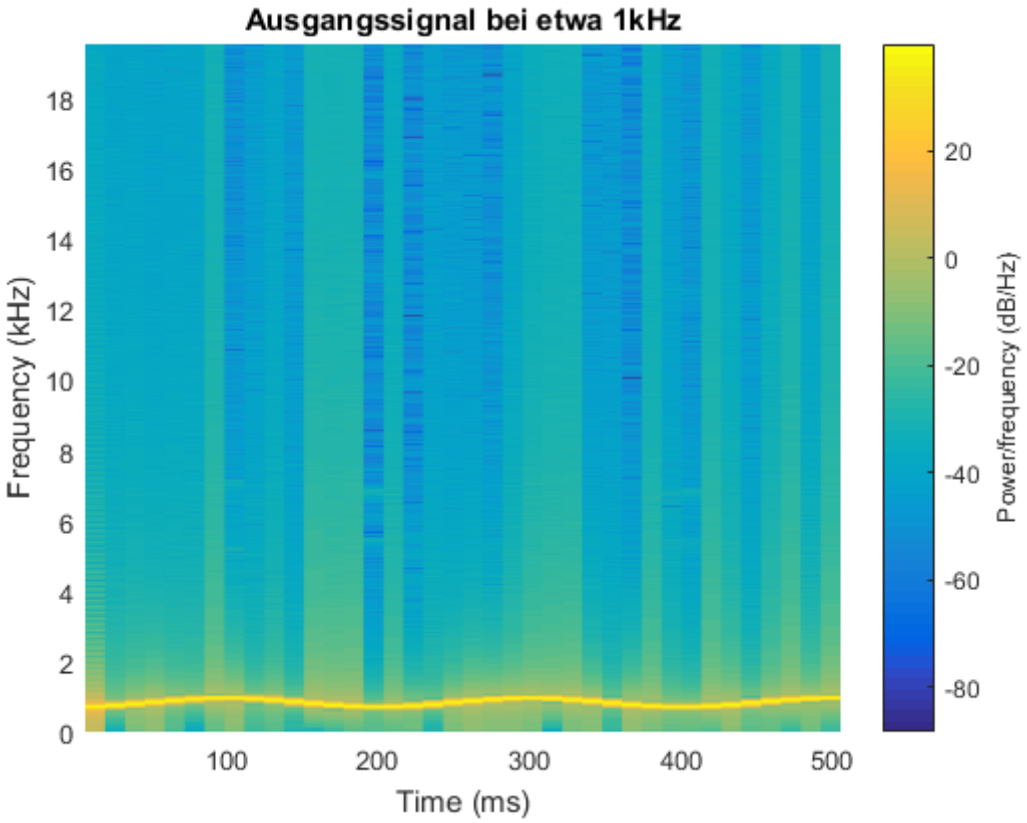


Abbildung 7.6.: Frequenzspektrum eines 1kHz Sinunssignals mit Vivado HLS)

In Abbildung 7.6 ist das Frequenzspektrum nach der Simulation mit Vivado HLS zu sehen. Der Ausgangssignal der C-Code Simulation entspricht in etwa der von der Simulation mit MATLAB in Abbildung 7.5.

Als nächstes können durch Direktiven der VHDL-Codesynthese angepasst werden. Für die

Kommunikation mit anderen VHDL-Modulen wird die Handshake-Schnittstelle des Entitys hinzugefügt. Des Weiteren wird für die for-Schleife die Pipeline-Schleife verwendet. Aus dem Bericht der Synthese kann gelesen werden, dass 405 Takte benötigt werden. Da zwischen zwei Abtastungen 2048 Takte sind, reicht die Zeit für die Berechnung.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	8	647	413
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	9	-	0	0
Multiplexer	-	-	-	8617
Register	-	-	6964	-
Total	9	9	7611	9030
Available	270	240	126800	63400
Utilization (%)	3	3	6	14

Abbildung 7.7.: Die benötigten Ressourcen für die Synthese des Vibrato VHDL-Codes

In der Abbildung 7.7 sind die benötigten Ressourcen im FPGA für das Synthetisieren des Vibrato-VHDL-Codes zu sehen.

7.4. Evaluation von HLS

In diesem Kapitel sollte die neue Software Vivado HLS von Xilinx getestet werden. Das Entwerfen eines selbstgeschriebenen VHDL-Codes für einen FIR-Filter hat gezeigt, dass eine einfache Addierstufe komplex werden kann. Von anderen Sprachen kommend, kann es ungewohnt sein mit parallelen Verarbeitung von Signalen in VHDL zu arbeiten und dies können zu Problemen und Fehlprogrammierung führen und somit beispielsweise zur kombinatorischen Schleifen führen. Dahingegen wird das prozedurale Programmieren mit C im Studium sehr früh eingeführt. Das kann genutzt werden um mit Vivado HLS aus einem C-Code ein VHDL-Code zu synthetisieren. Mit Vivado HLS kann einfache C-Codes leicht in VHDL synthetisiert werden. Weiterhin können durch Vivado-Eigene Bibliotheken Signale anpassen und optimieren. Die Verwendung von hls-math.h Bibliothek hat jedoch gezeigt, dass speziell die `sin`-Funktion bei der Synthese zu Problemen geführt hat. Zudem führen komplexen Berechnungen für die Synthese zur lange Rechenzeiten.

Leider hat das selbstgeschriebene VHDL-Code für den FIR-Filter nicht Funktioniert. Da die selben Koeffizienten für den Versuch mit HLS verwendet wurde, liegt das Problem nicht an den Koeffizienten. Wohl möglich liegt das Problem im VHDL-Code. Es wurde eine falsche Zuweisung getätigt.

8. Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung einer Audiocodec-Platine für das Digitaltechniklabor der Hochschule für Angewandte Wissenschaften (HAW) in Hamburg. Dabei soll die Platine durch einen FPGA gesteuert werden. Der FPGA soll zusätzlich das Audiosignal verarbeiten können. Abschließend sollte die Vivado High-Level Synthese Software von Xilinx testet werden.

Als erstes wurden einige notwendige Grundlagen erklärt und hergeleitet. Dazu gehören einige Grundlagen zur analoge Verstärkerschaltungen von Operationsverstärker sowie digitale Schaltungen. Weiterhin wurden einige Grundlagen zur Tontechnik, der Signalverarbeitung und der ModSys-Baseboard 2 beschrieben.

Danach wurden die Anforderungen an die Arbeit erörtert. Dazu gehörte die Anforderungen an die Auswahl des Audiocodec-Bausteins, an die Platine, an die Modellierung des VHDL-Codes zur Codeansteuerung sowie an die Bearbeitung mit dem Vivado HLS.

Anschließend wurde auf die Anforderungen aufbauend die Konzepte entwickelt. Als erstes wurde das Gesamtsystem betrachtet. Anschließend wurde daraus der Audiocodec-Baustein ausgewählt und kurz analysiert. Mit dem Baustein wurde ein Konzept für die Beschaltung sowie die Platinenentwurf gebildet. Anschließend wurde ein Konzept der Basisstruktur vom VHDL-Modell zur Steuerung des Audiocodec sowie die Signalbearbeitung entwickelt. Zum Schluss wurde ein Konzept zum Testen der Vivado HLS Software entwickelt.

Als nächstes wurde die analoge Schaltung für die Audiocodec-Platine samt Verstärkerschaltung nach den Anforderungen entwickelt. Nach der Bestückung der Platine wurde sie mit Spannung versorgt und die einzelnen Bausteine auf ihre Spannungspegel erfolgreich überprüft.

Der VHDL-Code für die Ansteuerung der Audiocodec-Platine wurde nach den Anforderungen geschrieben. Dabei wurde der VHDL-Code in kleine Module aufgeteilt, sodass jeder Teil eine kleine Teilaufgabe erfüllen konnte. Zudem wurde darauf geachtet, dass die Module in zwei Pfade aufgeteilt wurden: in Steuerpfad und in Signalpfad. Des Weiterem wurde jedes Modul mit dem selben Grundtakt versorgt. Mit der Audiocodec-Platine konnte somit die korrekte Funktionalität sowohl von der Audiocodec-Platine, als auch vom VHDL-Code demonstriert werden.

Zum Schluss sollten Erfahrungen mit der Vivado High-Level Synthese Software von Xilinx gemacht werden. Ein FIR-Filter wurde dafür entworfen und ein VHDL-Code wurde geschrieben. Mit dem selben FIR-Filterparameter wurde ein C-Code geschrieben und mit Vivado HLS ein VHDL synthetisiert und sollte miteinander verglichen werden. Jedoch kam es zur Komplikationen mit dem selbstgeschrieben FIR-Filter, sodass ein abschließender Vergleich nicht aussagekräftig war. In Anschluss wurde ein der Vibrato-Audioeffekt in MATLAB entworfen, in C übertragen und anschließend in VHDL Synthetisiert und getestet.

Literaturverzeichnis

[1] Texas Instruments Inc. *PCM3006; 16-Bit, Single-Ended Analog Input/Output Stereo Audio Codec*, 2000.

[2] Jürgen Reichardt. *Digitaltechnik, eine Einführung mit VHDL*, volume 4. Oldenburg, 2017.

[3] Wikipedia. Finite impulse response. URL: https://en.wikipedia.org/wiki/Finite_impulse_response letzter Zugriff am 16.06.2018.

[4] Udo Zölzer, editor. *DAFX Digital Audio Effects*. Wiley, 2011.

[5] Steve Trimberger. Die vier zeitalter der fpga-entwicklung, 2016. URL: https://www.elektronikpraxis.vogel.de/die-vier-zeitalter-der-fpga-entwicklung-a-541059/ letzter Zugriff am 16.06.2018.

[6] Steve Carr. Fpgas und programmierbare socs: Grundlagen und vorteile, 2017. URL: https://www.elektronikpraxis.vogel.de/grundzuege-und-vorteile-von-fpgas-und-programmierbaren-socs-a-66233 letzter Zugriff am 16.06.2018.

[7] Christian Siemers. Fpga oder mikrocontroller?, 2018. URL: https://www.elektronikpraxis.vogel.de/fpga-oder-mikrocontroller-a-505066/ letzter Zugriff am 16.06.2018.

[8] Walt Jung, editor. *Op Amp Applications Handbook*. Newnes, 2004.

[9] Vol. *Modulares System: Technische Beschreibung*, 10 2011. URL: https://www.haw-hamburg.de/fileadmin/user_upload/TI-IE/Daten/Labore/Digitaltechnik/UCF-Dateien/ModSys/BeschreibungModularesSystem.pdf letzter Zugriff am 16.06.2018.

[10] Xilinx. *7 Series FPGAs Data Sheet: Overview*, 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf letzter Zugriff am 16.06.2018.

[11] Jan Kumann Antti Lukats. *TE0725 TRM*, 2016. URL: https://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/3.5x7.3/TE0725/REV02/Documents/TRM-TE0725-02.pdf letzter Zugriff am 16.06.2018.

[12] Bernd Schwarz Jürgen Reichardt. *VHDL-Synthese*, volume 7. Oldenburg, 2015.

[13] *Die Frequenzskala*, 2017. URL: http://www.dasgesundeohr.de/ohr/304_die_frequenzskala.shtml letzter Zugriff am 16.06.2018.

[14] Andreas Friesecke. *Die Audio-Enzyklopädie*. De Gruyter Saur, 2014.

[15] Texas Instruments Inc. *SN74HC590A, 8-Bit Binary Counters with 3-State Output Registers*, 2003.

[16] Hank Zumbahlen, editor. *Linear Circuit Design Handbook*. Newnes, 2008.

[17] Analog Devices. *Audio Switching Amplifier*, 2017. URL: http://www.analog.com/media/en/technical-documentation/data-sheets/AD1990.pdf letzter Zugriff am 16.06.2018.

[18] LLC Semiconductor Components Industries. *LMV321 / LMV358 / LMV324 General-Purpose, Low Voltage, Rail-to-Rail Output Amplifiers*, 2002. URL: https://www.mouser.de/datasheet/2/308/LMV321-LMV358-LMV324-D-96261.pdf letzter Zugriff am 16.06.2018.

[19] Thomas Görne. *Tontechnik*. Carl Hanser Verlag, 2006.

[20] Xilinx, Inc. *Vivado Design Suite User Guide, High-Level Synthesis*, v2017.1 edition, May 2017.

[21] Xilinx, Inc. *Vivado Design Suite Tutorial, High-Level Synthesis*, v2017.2 edition, June 2017.

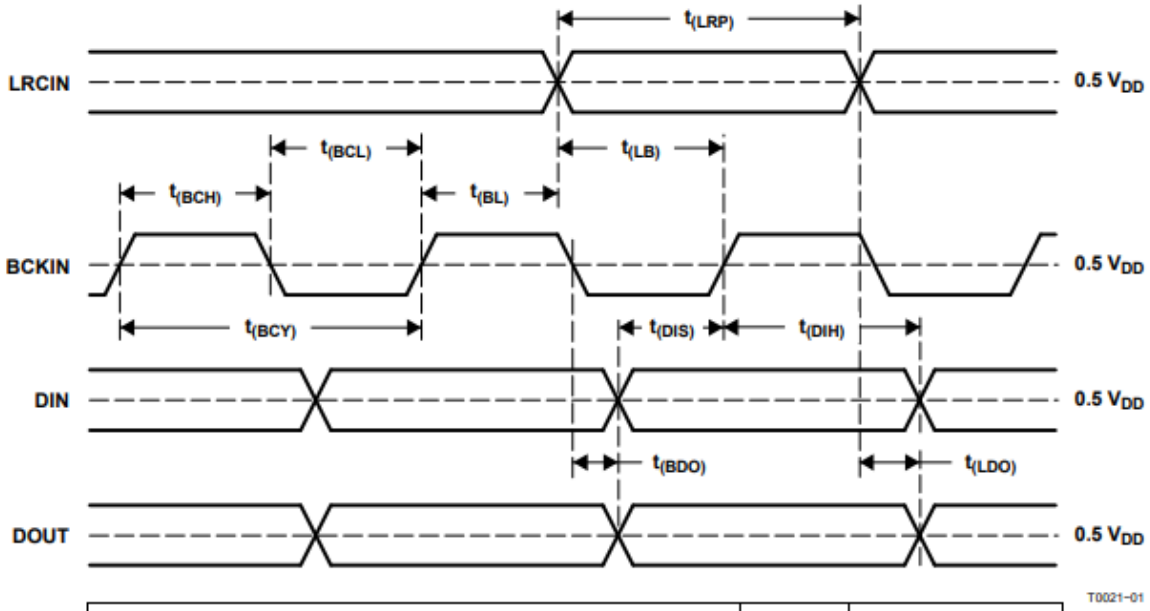
[22] Xilinx, Inc. *Loop Pipelining and Loop Unrolling*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html letzter Zugriff am 16.06.2018.

[23] Rulph Chassaing. *DSP Applications Using C and the TMS320C6x DSK*. John Wiley Sons, INC., 2002.

[24] Prof. Dr.-Ing. Bernd Schwarz. *FIR-Filterstrukturen und VHDL Implementierung mit Pipelining*. URL: http://users.etech.haw-hamburg.de/users/Schwarz/En/Lecture/Dyj/Notes/Chap4.pdf letzter Zugriff am 16.06.2018.

[25] Kupke Reichardt, Sauvagerd. *FIR-Filter Implementierung in MATLAB und in C*, 2012.

A.
Timingdiagramm von PCM3006



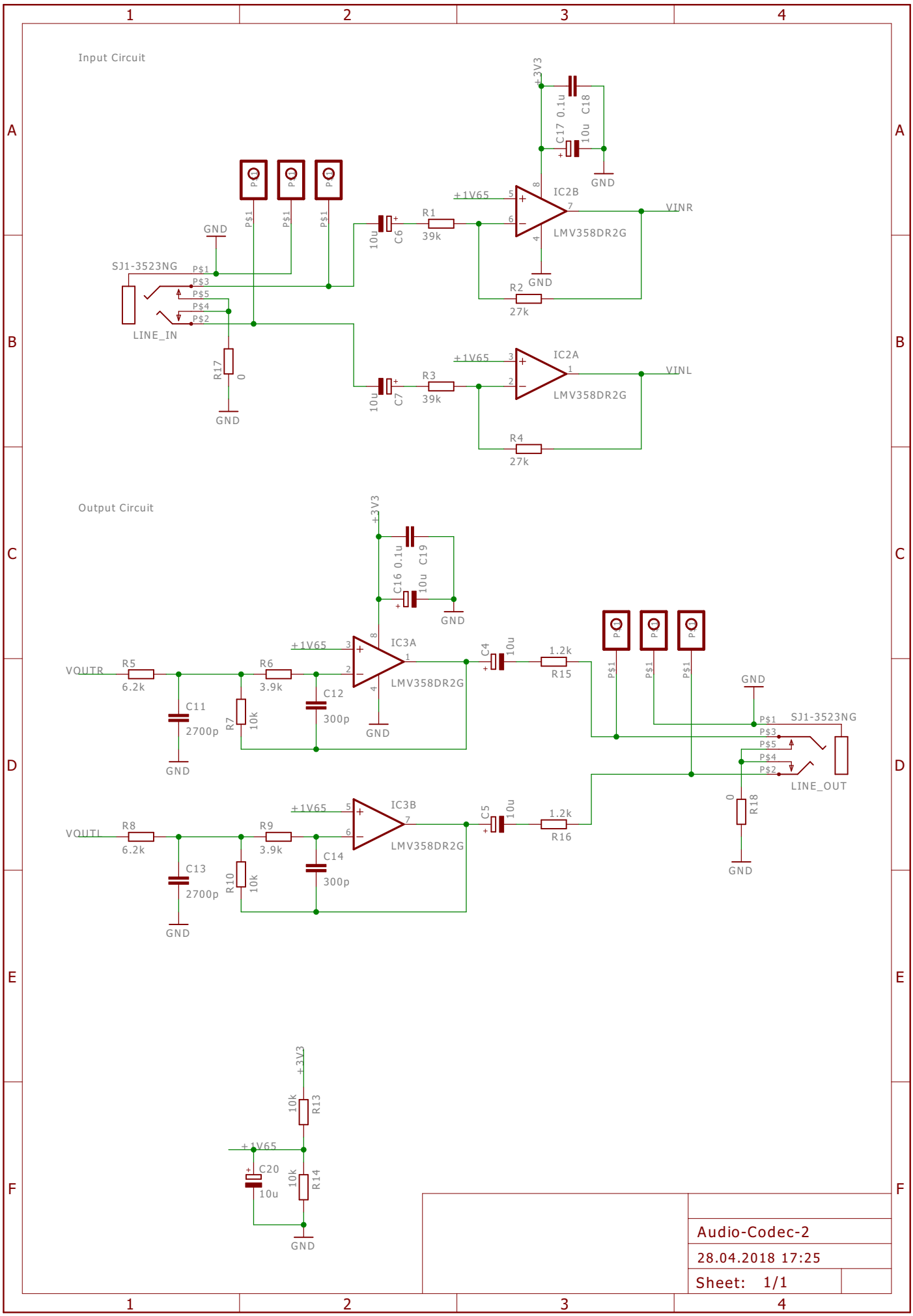
BCKIN pulse cycle time	$t_{(BCY)}$	300 ns (min)
BCKIN pulse duration, HIGH	$t_{(BCH)}$	120 ns (min)
BCKIN pulse duration, LOW	$t_{(BCL)}$	120 ns (min)
BCKIN rising edge to LRCIN edge	$t_{(BL)}$	40 ns (min)
LRCIN edge to BCKIN rising edge	$t_{(LB)}$	40 ns (min)
LRCIN pulse duration	$t_{(LRP)}$	$t_{(BCY)}$ (min)
DIN setup time	$t_{(DIS)}$	40 ns (min)
DIN hold time	$t_{(DIH)}$	40 ns (min)
DOUT delay time to BCKIN falling edge	$t_{(BDO)}$	40 ns (max)
DOUT delay time to LRCIN edge	$t_{(LDO)}$	40 ns (max)
Rising time of all signals	$t_{(RISE)}$	20 ns (max)
Falling time of all signals	$t_{(FALL)}$	20 ns (max)

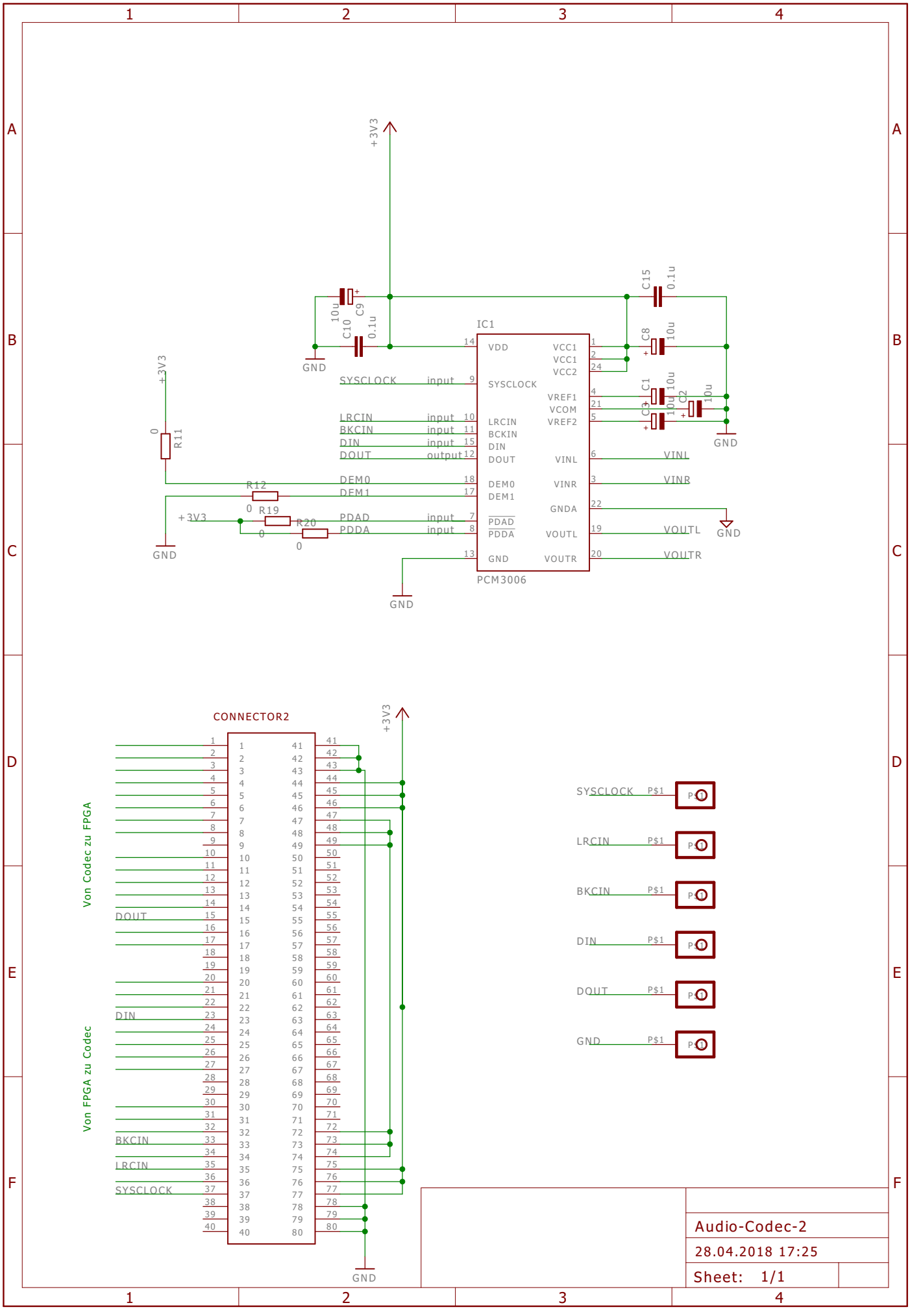
Abbildung A.1.: Timing-Diagramm von PCM3006, zeitliche AbhÃngigkeiten der erzeugten Signale zueinander mit spezifischer Dauer, die eingehalten werden mÃ¼ssen. [1]

B.
Analoge und digitale Schaltungen

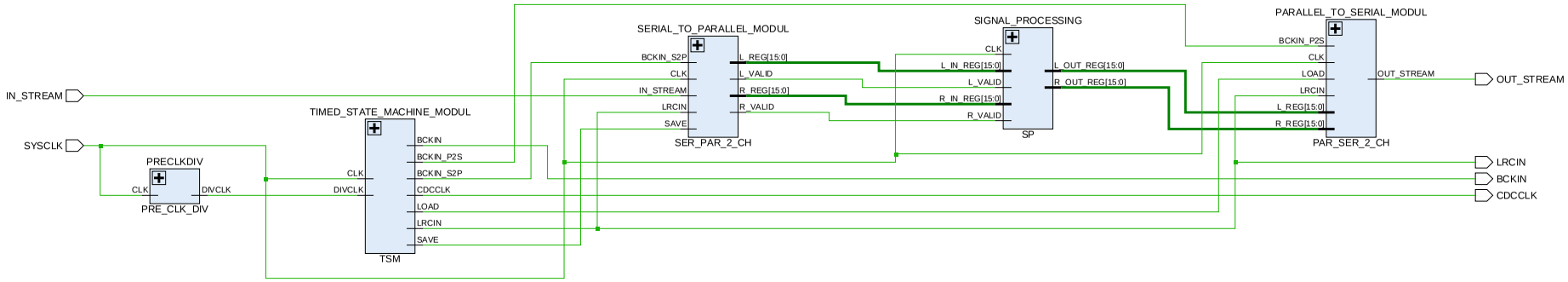
Hier wird das Layout der Platine sowie die Register-Transfer-Logik-Sicht der erstellten VHDL-Gesamtcode angezeigt. Auf dem CD befindet sich das EAGLE-Projekt von der Audiocodec-Platine. Auf Nachfrage erhÃltlich bei Herrn Prof. Dr. Lutz Leutelt.

B.1. Audiocodec Board Layout


Input Circuit
Output Circuit
+3V3
GND
+1V65
IC2B
IC2A
IC3A
IC3B
LMV358DR2G
SJ1-3523NG
LINE_IN
LINE_OUT
VINR
VINL
VOUTR
VOUTL
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R13
R14
R15
R16
R17
R18
C4
C5
C6
C7
C11
C12
C13
C14
C16
C17
C18
C19
C20
39k
27k
6.2k
3.9k
10k
1.2k
2700p
300p
0.1u
10u
0
Audio-Codec-2
28.04.2018 17:25
Sheet: 1/1


+3V3
C15
0.1u
C10
0.1u
C9
10u
GND
IC1
VDD
VCC1
VCC1
VCC2
C8
10u
SYSCLOCK
input
SYSCLOCK
VREF1
VCOM
VREF2
C1
C2
C3
10u
LRCIN
input
BKCIN
input
DIN
input
DOUT
output
LRCIN
BCKIN
DIN
DOUT
VINL
VINL
R11
0
R12
0
DEM0
DEM1
DEM0
DEM1
VINR
VINR
R19
0
R20
0
PDAD
input
PDDA
input
PDAD
PDDA
GNDA
VOUTL
VOUTL
GND
VOUTR
VOUTR
PCM3006
CONNECTOR2
Von Codec zu FPGA
Von FPGA zu Codec
DOUT
DIN
BKCIN
LRCIN
SYSCLOCK
SYSCLOCK
LRCIN
BKCIN
DIN
DOUT
GND
P$1
Audio-Codec-2
28.04.2018 17:25
Sheet: 1/1

B.2. RTL-Sicht der Basisstruktur


IN_STREAM
SYSCLK
PRECLKDIV
CLK
DIVCLK
PRE_CLK_DIV
TIMED_STATE_MACHINE_MODUL
CLK
DIVCLK
BCKIN
BCKIN_P2S
BCKIN_S2P
CDCCLK
LOAD
LRCIN
SAVE
TSM
SERIAL_TO_PARALLEL_MODUL
BCKIN_S2P
CLK
IN_STREAM
LRCIN
SAVE
L_REG[15:0]
L_VALID
R_REG[15:0]
R_VALID
SER_PAR_2_CH
SIGNAL_PROCESSING
CLK
L_IN_REG[15:0]
L_VALID
R_IN_REG[15:0]
R_VALID
L_OUT_REG[15:0]
R_OUT_REG[15:0]
SP
PARALLEL_TO_SERIAL_MODUL
BCKIN_P2S
CLK
LOAD
LRCIN
L_REG[15:0]
R_REG[15:0]
OUT_STREAM
PAR_SER_2_CH
OUT_STREAM
LRCIN
BCKIN
CDCCLK

C.
Codes Listings

In diesem Anhang befinden sich die VHDL-Codes. Weitere MATLAB, C und VHDL-Codes, sowie die Projektverzeichnisse von Vivado und Vivado HLS befinden sich CD. Auf Nachfrage erhÃltlich bei Herrn Prof. Dr. Lutz Leutelt.

```
1  -- Pre clock divider
2
   -- Teilt 80 MHz in 10 MHz
4  --

6  library ieee;
   use ieee.std_logic_1164.all;
8  use ieee.std_logic_unsigned.all;
   use ieee.numeric_std.all;
10
   entity PRE_CLK_DIV is
12 port(
           CLK: in std_logic;
14
           DIVCLK: out std_logic
16         );
   end PRE_CLK_DIV;
18
   architecture BEH_PRE_CLK_DIV of PRE_CLK_DIV is
20
   signal Q, QN: unsigned (1 downto 0):= (others=>'0');
22
   begin
24
   process(CLK)
26 begin
           if CLK='1' and CLK'event then
28                 Q <= QN after 2 ns;
           end if;
30 end process;

32 process(Q)
   begin
34         if Q = 3 then
                   QN <= (others=>'0') after 2 ns;
36                 DIVCLK <= '1' after 2 ns;
           else
38                 QN <= Q + 1 after 2 ns;
                   DIVCLK <= '0' after 2 ns;
40         end if;
   end process;
42
   end BEH_PRE_CLK_DIV;
```

```
1  library ieee;
   use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
   use ieee.numeric_std.all;
5
   entity TSM is
7  port(
           CLK: in std_logic;
9          DIVCLK: in std_logic;

11         BCKIN: out std_logic;
           LRCIN: out std_logic;
13         CDCCLK: out std_logic;

15         BCKIN_S2P: out std_logic;
           BCKIN_P2S: out std_logic;
17         SAVE: out std_logic;
           LOAD: out std_logic
19         );
   end TSM;
21
   architecture BEH_TSM of TSM is
23
   signal INP_BCKIN_1, INP_BCKIN_2: std_logic;
25 signal INP_LRCIN_1, INP_LRCIN_2: std_logic;

27 signal Q_DIV, QN_DIV: unsigned( 8 downto 0):= (others=>'0');

29 begin

31 -- Zähler zur Generierung von LRCIN, BCKIN und CDCCLK

33 process(CLK)
   begin
35         if CLK='1' and CLK'event then
                   if DIVCLK = '1' then
37                         Q_DIV <= QN_DIV after 2 ns;
                   end if;
39         end if;
   end process;
41
   process(Q_DIV)
43 begin
```

```
        QN_DIV <= Q_DIV + 1 after 2 ns;
end process;

CDCCLK <= Q_DIV(0) after 2 ns;
BCKIN <= Q_DIV(3) after 2 ns;
LRCIN <= Q_DIV(8) after 2 ns;


PULSE_SHORTER: process(CLK)
begin
        if CLK='1' and CLK'event then
                INP_BCKIN_1 <= Q_DIV(3) after 2 ns;
                INP_BCKIN_2 <= INP_BCKIN_1 after 2 ns;

                INP_LRCIN_1 <= Q_DIV(8) after 2 ns;
                INP_LRCIN_2 <= INP_LRCIN_1 after 2 ns;
        end if;
end process PULSE_SHORTER;

BCKIN_S2P <= INP_BCKIN_1 and (not INP_BCKIN_2) after 2 ns;
BCKIN_P2S <= INP_BCKIN_2 and (not INP_BCKIN_1) after 2 ns;

SAVE <= (INP_LRCIN_1 and (not INP_LRCIN_2)) or (INP_LRCIN_2 and (
   not INP_LRCIN_1)) after 2 ns;
LOAD <= (INP_LRCIN_1 and (not INP_LRCIN_2)) or (INP_LRCIN_2 and (
   not INP_LRCIN_1)) after 2 ns;

end BEH_TSM;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;

6  entity SER_PAR is
   port(
8          CLK: in std_logic;
           EN: in std_logic;
10         BCKIN: in std_logic;  -- BITCLK_EN
           IN_STREAM: in std_logic;
12         SAVE: in std_logic;

14         VALID: out std_logic;
           OUT_REG: out std_logic_vector(15 downto 0)
16         );
   end SER_PAR;
18
   architecture BEH_SER_PAR of SER_PAR is
20
   signal INT_REG: std_logic_vector( 15 downto 0);
22
   begin
24
   process( CLK)
26 begin
           if CLK = '1' and CLK'event then
28                 if (EN = '1') and (BCKIN = '1') then
                           INT_REG <= INT_REG(14 downto 0) &
                              IN_STREAM after 2 ns;
30                 end if;
                   VALID <= '0' after 2 ns;
32                 if SAVE = '1' then
                           OUT_REG <= INT_REG after 2 ns;
34                         VALID <= '1' after 2 ns;
                   end if;
36         end if;
   end process;
38
   end BEH_SER_PAR;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;

6  entity SER_PAR_2_CH is
   port(
8          CLK: in std_logic;
           LRCIN: in std_logic;
10         BCKIN_S2P: in std_logic;
           IN_STREAM: in std_logic;
12         SAVE: in std_logic;

14         L_VALID, R_VALID: out std_logic;
           L_REG, R_REG: out std_logic_vector( 15 downto 0)
16         );
   end SER_PAR_2_CH;
18
   architecture BEH_SER_PAR_2_CH of SER_PAR_2_CH is
20
   signal L_SAVE, R_SAVE: std_logic;               -- intern save
      signal
22 signal N_EN: std_logic;                  -- negierter LRCIN signal

24 begin

26 N_EN <= not LRCIN after 2 ns;
   L_SAVE <= SAVE and not LRCIN after 2 ns;
28 R_SAVE <= SAVE and LRCIN after 2 ns;

30
   -- Linker Kanal
32 L_SERPAR: entity work.SER_PAR
           port map(CLK, LRCIN, BCKIN_S2P, IN_STREAM, L_SAVE, L_VALID
              , L_REG);
34
   -- Rechter Kanal
36 R_SERPAR: entity work.SER_PAR
           port map(CLK,  N_EN, BCKIN_S2P, IN_STREAM, R_SAVE, R_VALID
              , R_REG);
38

40 end BEH_SER_PAR_2_CH;
```

```
1  -- Output Shifting Module
2  --
   -- Dieses Modul wandelt zwei 16 Bit Register in ein 1-Bit Strom um
       . Es
4  -- handelt um zwei KanÃle. Angepasst an PCM3006
   --
6  -- Wortwechsel RL: Rechts = 0; Links = 1

8
   library ieee;
10 use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
12 use ieee.numeric_std.all;

14 entity PAR_SER is
   port(
16         CLK: in std_logic;                          -- Systemtakt
           EN: in std_logic;                 -- Modul an und
              ausschalten
18         BCKIN: in std_logic;
           LOAD: in std_logic;
20         IN_REG: in std_logic_vector(15 downto 0);        --
              Register

22         OUT_STREAM: out std_logic
           );
24 end PAR_SER;

26 architecture BEH_PAR_SER of PAR_SER is

28 signal INT_REG: std_logic_vector(15 downto 0);

30 begin

32 ------------SHIFTING---------------------------

34 SHIFTING: process( CLK)
   begin
36         if CLK='1' and CLK'event then
                   if LOAD ='1' then
38                         INT_REG <= IN_REG after 2 ns;
                   elsif BCKIN='1' then -- afternativ: if BCKIN='1'
                      and EN='1' then
```

```
40                          INT_REG <= INT_REG(14 downto 0) & '0'
                               after 2 ns;
                end if;
42          end if;
end process SHIFTING;
44
OUT_STREAM <= INT_REG(15)  after 2 ns;
46

48 end BEH_PAR_SER;
```

```
1  library ieee;
   use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
   use ieee.numeric_std.all;
5
   entity PAR_SER_2_CH is
7  port(
           CLK: in std_logic;
9          LRCIN: in std_logic;
           BCKIN_P2S: in std_logic;
11         LOAD: in std_logic;
           L_REG: in std_logic_vector( 15 downto 0);
13         R_REG: in std_logic_vector( 15 downto 0);

15         OUT_STREAM: out std_logic
           );
17 end PAR_SER_2_CH;

19 architecture BEH_PAR_SER_2_CH of PAR_SER_2_CH is

21 --signal IN_REG: std_logic_vector( 15 downto 0);
   signal OUT_STREAM_R, OUT_STREAM_L: std_logic;
23
   begin
25
   OUT_STREAM <= OUT_STREAM_L when LRCIN = '1' else OUT_STREAM_R
      after 2 ns;
27
   -- beide KanÃle
29 PARSER_L: entity work.PAR_SER
           port map(CLK, LRCIN, BCKIN_P2S, LOAD, L_REG, OUT_STREAM_L)
              ;
31
   PARSER_R: entity work.PAR_SER
33     port map(CLK, LRCIN, BCKIN_P2S, LOAD, R_REG, OUT_STREAM_R);

35 end BEH_PAR_SER_2_CH;
```

```
1  -- CODEC Monitor
2
   library ieee;
4  use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
6
   entity CM is
8  port(
           SYSCLK: in std_logic;
10         IN_STREAM: in std_logic;

12         CDCCLK, BCKIN, LRCIN: out std_logic;
           OUT_STREAM: out std_logic
14         );
   end CM;
16
   architecture BEH_CM of CM is
18
   signal DIVCLK: std_logic;
20 signal INT_BCKIN: std_logic;
   signal INT_LRCIN: std_logic;
22 signal BCKIN_S2P: std_logic;
   signal BCKIN_P2S: std_logic;
24 signal SAVE: std_logic;
   signal LOAD: std_logic;
26 signal L_VALID, R_VALID: std_logic;

28 signal L_IN_REG, R_IN_REG: std_logic_vector(15 downto 0);
   signal L_OUT_REG, R_OUT_REG: std_logic_vector(15 downto 0);
30
   begin
32
   BCKIN <= INT_BCKIN;
34 LRCIN <= INT_LRCIN;

36
   PRECLKDIV: entity work.PRE_CLK_DIV
38         port map( SYSCLK, DIVCLK);

40 TIMED_STATE_MACHINE_MODUL: entity work.TSM
           port map( SYSCLK, DIVCLK, INT_BCKIN, INT_LRCIN, CDCCLK ,
              BCKIN_S2P, BCKIN_P2S, SAVE, LOAD);
42
```

```
SERIAL_TO_PARALLEL_MODUL: entity work.SER_PAR_2_CH
        port map( SYSCLK, INT_LRCIN, BCKIN_S2P, IN_STREAM, SAVE,
           L_VALID, R_VALID, L_IN_REG, R_IN_REG);

PARALLEL_TO_SERIAL_MODUL: entity work.PAR_SER_2_CH
        port map( SYSCLK, INT_LRCIN, BCKIN_P2S, LOAD, L_OUT_REG,
           R_OUT_REG, OUT_STREAM);

SIGNAL_PROCESSING: entity work.SP
        port map( SYSCLK, L_VALID, R_VALID, L_IN_REG, R_IN_REG,
           L_OUT_REG, R_OUT_REG);

end BEH_CM;
```

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 22. Juni 2018	
Ort, Datum	Unterschrift