Baccalaureatsarbeit
Entwicklung eines Ortungsgeräts für Blinde

T. Knepper B. Poettering

13. Juni 2002

Ziel diese Projekts war ein FPGA-Entwurf eines Ortungsgeräts für Blinde. Beteiligt an der Realisierung waren P. Grotrian, T. Knepper, B. Poettering und F. Rinneberg.

Die folgenden Kapitel wurden von T. Knepper geschrieben:

- Kapitel: 2.1
- Kapitel: 2.2
- Kapitel: 2.3
- Kapitel: 3.2
- Kapitel: 4

Die folgenden Kapitel wurden von B. Poettering geschrieben:

- Kapitel: 1
- Kapitel: 2.4
- Kapitel: 3.1

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Übersicht

Ziel des Projekts ist es, ein Ortungsgerät für Blinde zu entwickeln, welches diesen ermöglichen soll, sich über akustische Signale eine Vorstellung ihrer räumlichen Umgebung machen zu können. Das fertige Gerät, z.B. als Helm getragen, soll per Ultraschall die Entfernungen zu Objekten im Raum messen und aus diesen einen Ton modulieren, der dem Träger Aufschluss über Position und Größe von Hindernissen gibt. Dabei wird ausgenutzt, dass der Mensch in der Lage ist, die in gehörten Audiosignalen überlagerten Signale nach Frequenzen zu trennen und einzeln zu interpretieren. Daraus ergibt sich die Möglichkeit, dass mehrere Entfernungen (zu verschiedenen Objekten) gleichzeitig gemessen und auf verschiedene Frequenzen abgebildet werden können, die auch gleichzeitig kognitiv erfasst werden können.

In Abbildung 1.1 wird ersichtlich, wie die Abbildung von Entfernungen auf Frequenzen aussehen könnte. Die gemessenen Entfernungen von 2 m bzw. 8 m werden durch Frequenzen von 500 Hz bzw. 2 kHz dargestellt.

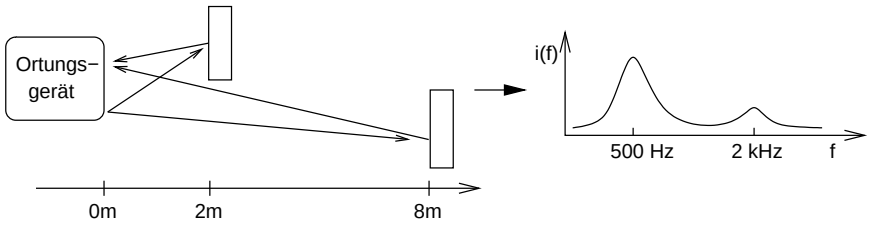


Abbildung 1.1: Mögliches Mapping von Entfernungen auf Frequenzen

1.2 Funktionsweise des Geräts

In Entfernungsmessern via Ultraschall (z.B. bei Abstandswarnern in Autos) werden in der Regel von einem Ultraschallsender Impulse (halb-)kugelförmig oder gerichtet abgestrahlt. Diese Impulse werden von Objekten reflektiert und von Ultraschallempfängern wieder aufgefangen. Aus den gemessenen Laufzeiten können dann die Entfernungen zu den reflektierenden Flächen bestimmt werden.

In der Praxis erweisen sich Ultraschall-Impulse zur präzisen Entfernungsmessung als unpraktikabel. Dies liegt daran, dass Impulse (durch ihre steilen Flanken) nicht bandbegrenzt sind und daher eine saubere Erkennung des reflektierten Signals durch die Ultraschallkapseln mit nur endlichem Spektrum unmöglich ist. Besser wäre es, wenn die Ultraschallquelle keine Impulse, sondern stattdessen sinus-förmige Signale abstrahlen würde. Bleibt deren Frequenz und Amplitude über die Zeit konstant (z.B. bei 40 kHz), lassen sich jedoch offensichtlich keine Laufzeitunterschiede mehr feststellen. Dieses Problem wird bei der Verwendung sog. Chirps vermieden.

Ein Ultraschall-Chirp ist ein Signal im Ultraschallbereich, das entsteht, wenn bei einem cosinus-förmigen Signal die Frequenz über der Zeit rampenförmig geändert wird. Parameter eines Chirps sind seine Start- und Endfrequenz, sowie die Chirpsteigung. Während einer Chirpdauer wird ein cosinus-Signal linear steigender Frequenz erzeugt. Chirpfolgen sind periodische Signale, die entstehen, wenn der cosinus statt durch eine Rampe durch einen Sägezahn moduliert wird. Abbildung 1.2 verdeutlicht das.

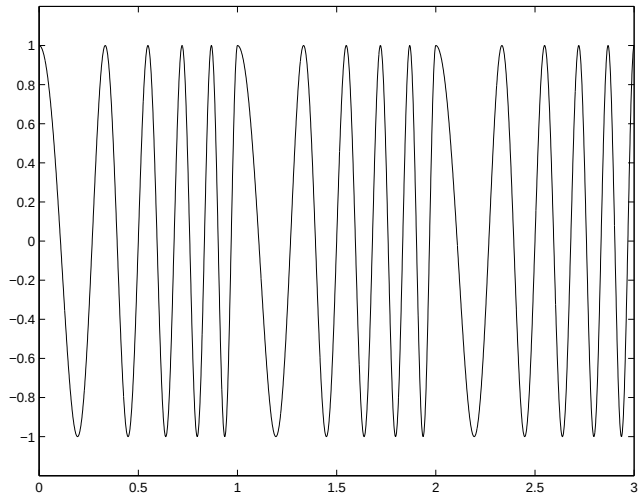


Abbildung 1.2: Eine Chirpfolge im Zeitbereich

Strahlt ein Ultraschall-Sender Chirps ab, kommt durch die unterschiedlichen Laufzeiten zu den reflektierenden Objekten beim Empfänger eine Überlagerung zueinander zeitverschobener Chirps an. Die Zeitversätze hängen linear von den jeweiligen Laufzeiten ab. Die genaue Position innerhalb eines empfangenen Chirps (also der Zeitversatz) wird durch die jeweils empfangene Frequenz charakterisiert. Also kann zu den aufgefangenen Chirps die jeweils zurückgelegte Entfernung eindeutig ermittelt werden.

1.3 Formale Definition

Ein möglicher Ansatz, um von den aufgenommenen Signalen auf die Entfernungsdaten zu schließen, wäre also die Ausführung einer Fourier-Transformation auf die einkommenden Daten. Die Entfernungen sind proportional zur **Differenz** zwischen empfangenem Frequenzanteil und gerade gesendeter Frequenz (siehe Abbildung 1.3).

Es gibt neben der Fouriertransformation noch eine effizientere Methode, um die Laufzeiten

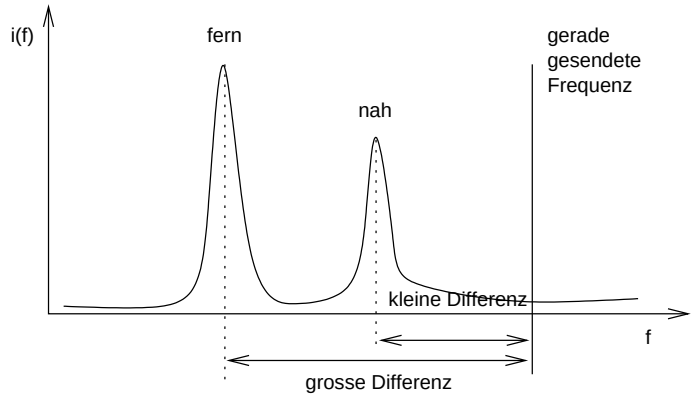


Abbildung 1.3: Interpretation aufgenommener Frequenzen bei der Verwendung von Chirps

der Chirps auf hörbare Frequenzen abzubilden: die Multiplikation der ein- und ausgehenden Signale. Dieses Verfahren lässt sich aus einem Additionstheorem für trigonometrische Funktionen herleiten:

$$\frac{1}{2}(\cos\alpha + \cos\beta) = \cos\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2}$$

Mit $\theta = \frac{\alpha+\beta}{2}$ und $\vartheta = \frac{\alpha-\beta}{2}$ (also $\alpha = \theta + \vartheta$ und $\beta = \theta - \vartheta$) gilt dann:

$$\cos\theta \cdot \cos\vartheta =$$

$$\cos(\frac{\alpha+\beta}{2})\cos(\frac{\alpha-\beta}{2}) =$$

$$\frac{1}{2}(\cos\alpha + \cos\beta) =$$

$$\frac{1}{2}(\cos(\theta+\vartheta) + \cos(\theta-\vartheta))$$

Generell gilt: wenn zwei cosinus-förmige Signale multipliziert werden, so setzt sich Produkt aus zwei cosinus-Signalen zusammen: das eine hat als Frequenz die Differenz der ursprünglichen Frequenzen, das andere die Summe der ursprünglichen Frequenzen.

Ein Chirp $c(t)$ kann formal als linear beschleunigte Drehung definiert werden:

$$c(t) = cos(\varphi_0 + \dot{\varphi}t + \frac{1}{2}\ddot{\varphi}t^2)$$

Vernachlässigt man die Phasenlage φ_0 und lässt ω der Startfrequenz des Chirps und $\psi = 2\pi\frac{df}{dt}$ der Chirpsteigung entsprechen, erhält man:

$$c(t) = cos(\omega t + \psi t^2)$$

Dann gilt für das Produkt von aus- und eingehendem Signal (wenn letzteres τ Zeiteinheiten unterwegs war):

$$c(t) \cdot c(t-\tau) =$$

$$cos(\omega t + \psi t^2) \cdot cos(\omega(t-\tau) + \psi(t-\tau)^2) =$$

$$\frac{1}{2}\cos(\omega t + \psi t^2 + \omega t - \omega\tau + \psi t^2 - 2\psi\tau t + \psi\tau^2) +$$

$$\frac{1}{2}\cos(\omega t + \psi t^2 - \omega t + \omega\tau - \psi t^2 + 2\psi\tau t - \psi\tau^2) =$$

$$\frac{1}{2}\left(\underbrace{\cos\left((-\omega\tau + \psi\tau^2) + 2(\omega - \psi\tau)t + 2\psi t^2\right)}_{B} + \underbrace{\cos\left((\omega\tau - \psi\tau^2) + 2\psi\tau t\right)}_{A}\right)$$

Eine Veranschaulichung des Ergebnisses ist in Abbildung 1.4 zu sehen. Wichtig ist im unteren Diagramm, dass $2\psi\tau$ (aus Term A) nicht von t abhängig ist, sondern proportional zur zurückgelegten Entfernung ist (die wiederum proportional zu τ ist). Das gilt zwar nicht für Term B, dessen Werte sind jedoch immer grösser als $2\omega - 2\psi\tau$, so dass sie bei geeigneter Wahl von ω und ψ oberhalb der Hörgrenze des Menschen (ca. 22 kHz) liegen und daher nicht mehr wahrgenommen werden.

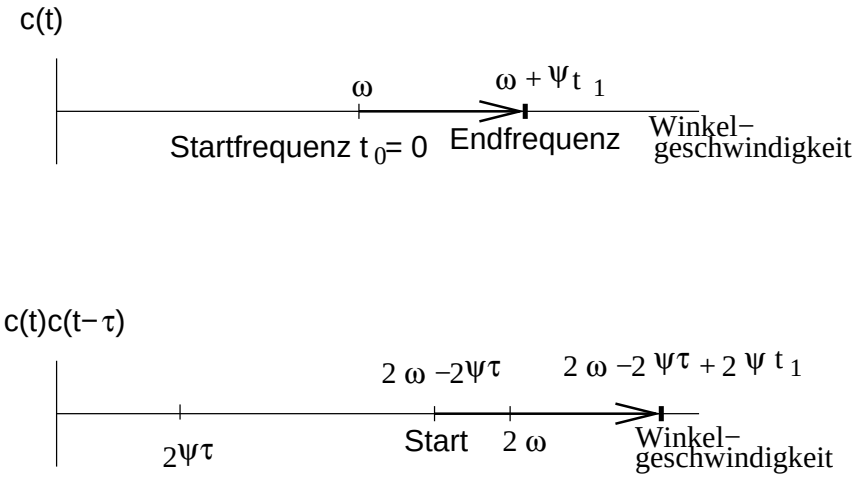


Abbildung 1.4: Veranschaulichung der Multiplikation zweier Chirps

Da Chirps nur von endlicher Länge sind und darum periodisch ausgestrahlt werden müssen, kommt es vor, dass während der Aussendung eines Chirps reflektierte Signale des letzten Chirps aufgefangen werden. Die Differenzfrequenz zwischen gerade gesendetem und gerade emfangenem Signal beträgt im Normalfall $\Delta f = 2\psi\tau$. Wird jedoch das Signal vom letzten Chirp aufgefangen, so ergibt sich $\Delta f^* = (f_2 - f_1) - \Delta f$ (siehe Abbildung 1.5), wobei hier f_1 bzw. f_2 die Start- bzw. Endfrequenzen eines Chirps sind sind. Bei periodischen Chirps treten also pro Weglänge zwei Frequenzen auf: sowohl Δf also auch Δf^*.

Die Frequenzen Δf^* sind lassen sich jedoch mit Hilfe eines Tiefpasses wegfiltern, sind sie bei geeigneter Wahl von f_1 und f_2 doch immer groesser als Δf. Nach dieser Filterung kann das Signal direkt auf den Lautsprecher ausgegeben werden, um einem Blinden zur Raumorientierung zu dienen. Der sich ergebende strukturelle Aufbau des Geräts ist in Abbildung 1.7 skizziert.

Es sind noch die Parameter des Chirps zu bestimmen: Startfrequenz, Zielfrequenz und seine Steigung. Im Versuch wurde bestimmt, dass handelsübliche Ultraschallsender/-empfänger

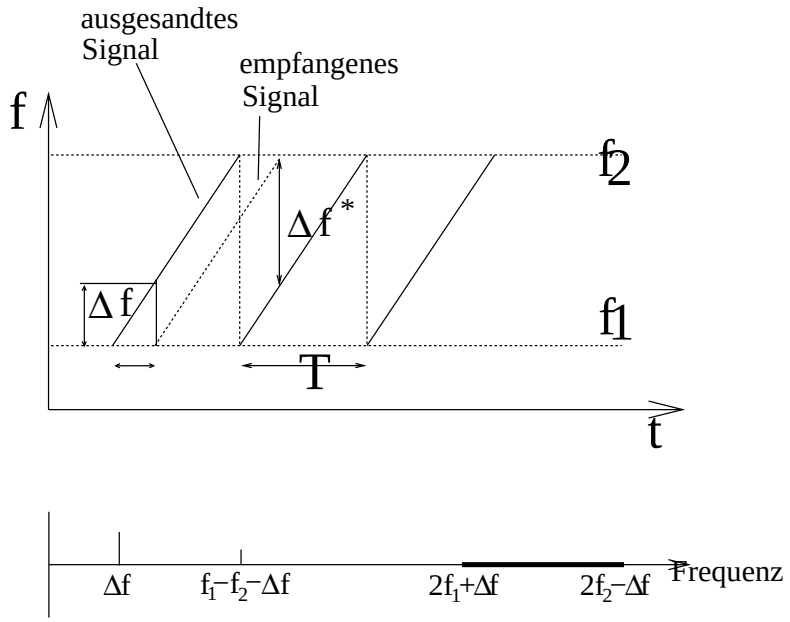


Abbildung 1.5: Resultate periodischer Chirps

in Resonanzfrequenz von ca. 41 kHz betrieben werden. Eine gemessene Kennlinie kann Abbildung 1.6 entnommen werden. Für die Anwendung ist es wichtig, dass die verwendeten Chirpfrequenzen nicht verschieden stark von den Ultraschallkapseln gedämpft werden. Wir entschieden uns für das Intervall $[45kHz, 52kHz]$, welches im Versuch bessere Ergebnisse lieferte als die Alternative $[30kHz, 35kHz]$. Ebenfalls durch Ausprobieren wurde eine Steigung von $35\frac{kHz}{sec}$ als geeignet ermittelt.

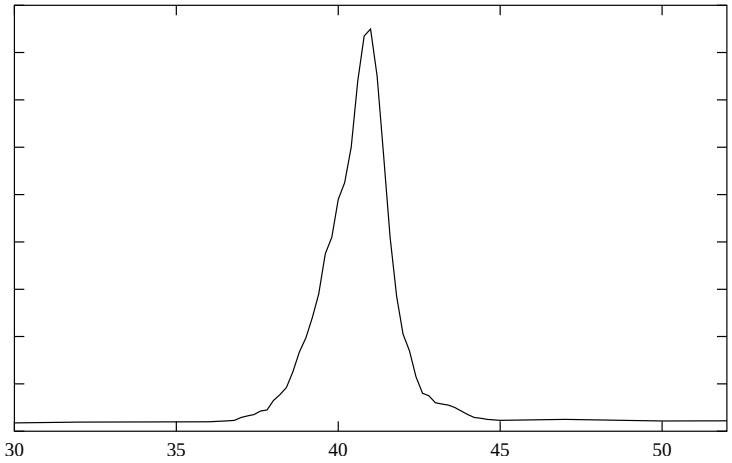


Abbildung 1.6: Das Resonanzverhalten handelsüblicher Ultraschallsender und -empfänger

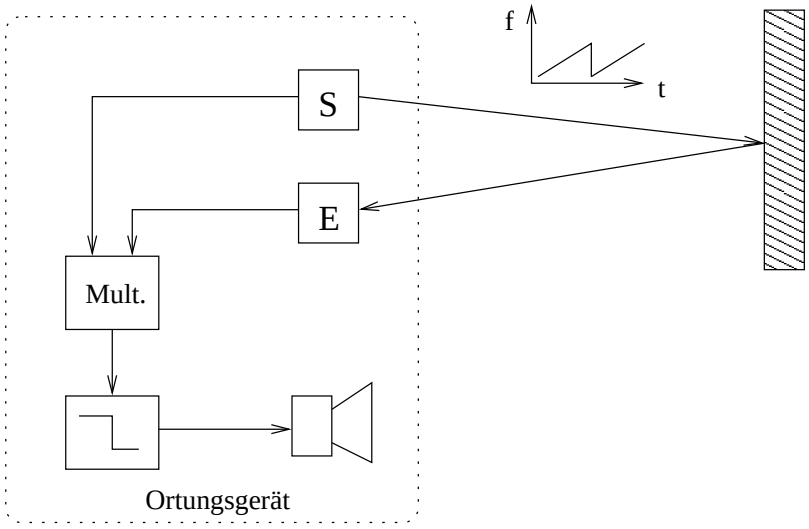


Abbildung 1.7: Struktureller Aufbau des Ortungsgeräts

Kapitel 2

Aufbau

Soll der Vorschlag von Abbildung 1.7 in Hardware gegossen werden, sind neben dem Multiplizierer und dem Filter noch weitere Komponenten notwendig. Diese sind in Abbildung 2.1 gezeigt.

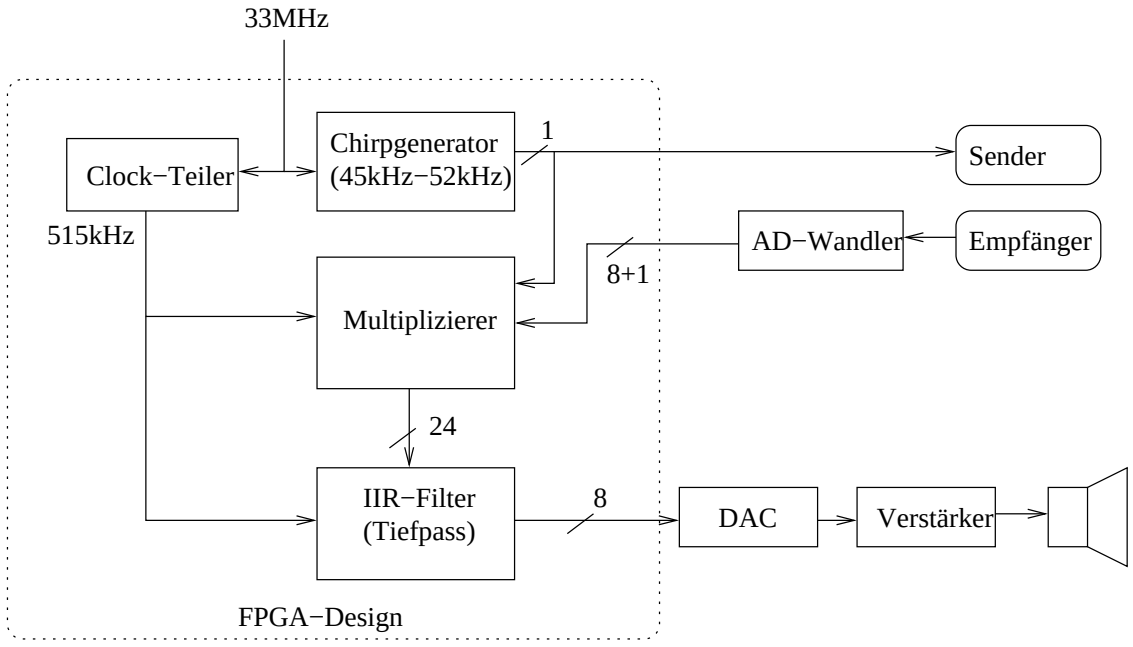


Abbildung 2.1: Komponenten des Ortungsgeräts im FPGA

- Der Clockteiler soll den Sampletakt für A/D- und D/A-Wandler zur Verfügung stellen. Dazu teilt er den Systemtakt (33 MHz) entsprechend herunter.
- Der Chirpgenerator generiert eine Chirpfolge als Ausgangssignal.
- Der Multiplizierer multipliziert diese Chirpfolge mit dem gerade empfangenem Signal.
- Der Tiefpass filtert hohe Frequenzanteile aus dem Resultat des Multiplizierers.
- Sender und Empfänger sind jeweils Ultraschallkapseln (incl. Vorverstärkern usw.).

- Der A/D-Wandler wandelt das empfangene Analogsignal in ein 8 Bit breites Digitalsignal um. Ein Überlauf-Bit wird getrennt herausgeführt.
- Der DAC wandelt digitale in analoge Signale um. Diese werden verstärkt und auf den Lautsprecher ausgegeben.

Die Schaltung wurde mit einem NIOS-Prototyping-Board der Firma Altera realisiert. Alle Komponenten innerhalb der gestrichelten Linie (Abbildung 2.1) sind in VHDL programmiert und können in das FPGA geladen werden. Die übrigen Komponenten sind durch externe Bauteile realisiert. Es ist darauf zu achten, dass die D/A- bzw. A/D-Wandler schnell genug für die verwendeten Samplefrequenzen sein müssen.

2.1 Der Frequenzteiler

Der Teiler ist eine Hilfskomponente, um den für die Signalverarbeitung notwendigen Takt von 515 kHz aus dem Systemtakt von 33 MHz zu erzeugen. Dazu werden die 33 MHz durch $2^6 = 64$ geteilt, indem ein 6-Bit breiter Zähler bei jedem Takt inkrementiert wird. Das oberste Bit wechselt dann genau mit der richtigen Frequenz.

Nach dem Nyquisttheorem muss die Samplingrate mindestens doppelt so gross sein wie die maximale Frequenz aller betroffenen Signale, um Aliaseffekte auszuschliessen. In unserem Fall bedeutet das, dass wir mit mindestens $2 \cdot 2 \cdot 52kHz = 208kHz$ sampeln müssen, da nach der Multiplikation Frequenzen bis $104kHz$ im Signal enthalten sind.

Als Folge entschieden wir uns für eine Abtastfrequenz von 515 kHz, obwohl 275 kHz ausgereichen, weil

- weniger Logikzellen auf dem FPGA gebraucht werden
- der Entwurf ausreichend schnell ist
- wir damit auf eine eventuelle Nutzung weiterer Frequenzbereiche vorbereitet sind.

2.2 Der Chirpgenerator

Als eine sehr wichtige Komponente im Design ist der Chirpgenerator für die Erzeugung des Signals und für die Synchronisation des restlichen Systems auf dieses Signal zuständig.

In dieser Komponente erzeugen wir ein Rechtecksignal, welches aber aufgrund der Tiefpass-Charakteristik der von uns verwendeten Sender-/Empfänger-Kombination[1] in einem sinusähnlich ausgesandten Signal resultiert. Bleiben noch die Auswirkung bei der Muliplikation. Wie in Abb. 2.2 und 2.3 zu sehen, entstehen bei der Multiplikation des ausgesandten Chirps mit einer Rechteckimpulsfolge nicht gewünschte Frequenzanteile. Diese können aber durch ein entsprechend designtes Filter ausreichend stark gedämpft werden.

Zur Generierung des Chirps (Abb: 2.4) wird zu einem 48 Bit breiten Akkumulator ein von der auszugebenden Frequenz abhängiger Offset bei jedem Takt der 33 MHz addiert. Das oberste Bit des Akkus entspricht dann direkt dem Sample des zu erzeugenden Signal. Damit wir eine steigende Frequenz erhalten, wird der Offset regelmässig bis zu seinem Maximum

[1] Ultraschallsender:$400ST$ Empfänger:$400SR$

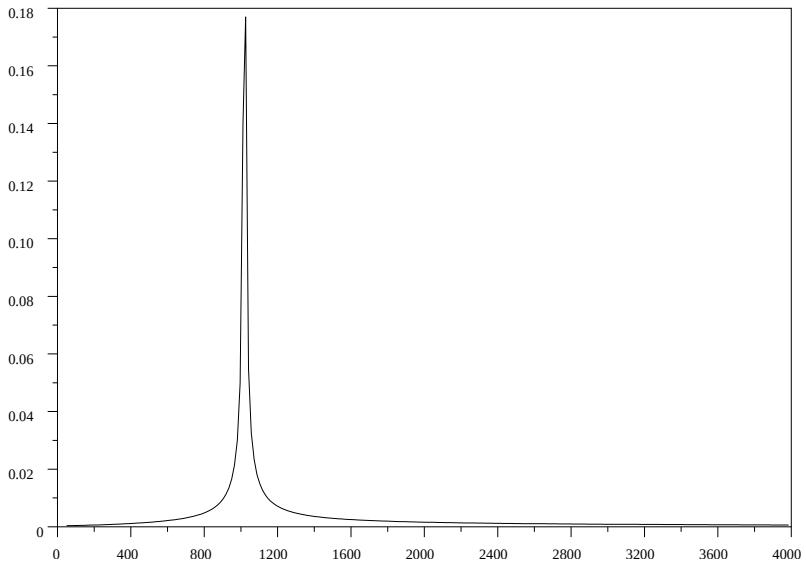


Abbildung 2.2: $\sin(46kHz) \cdot \sin(47kHz)$

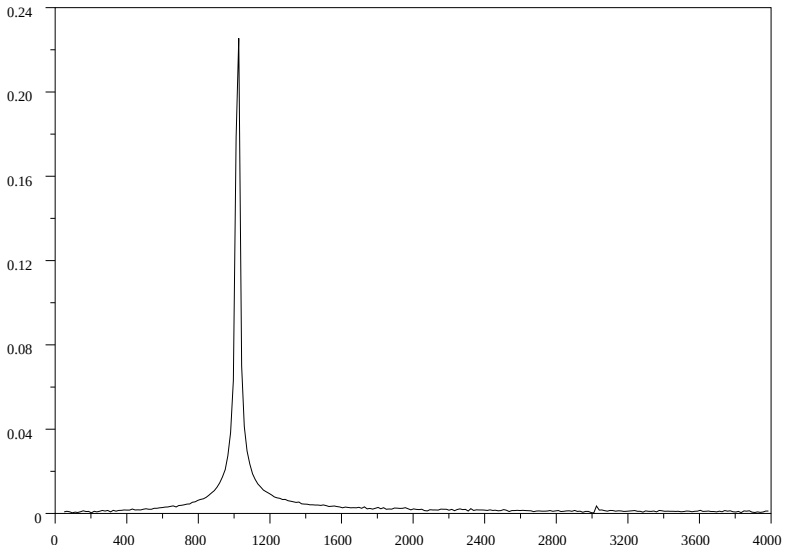


Abbildung 2.3: Multiplikation einer Sinusschwingung mit einer Rechteckimpulsfolge

inkrementiert. Die Parameter sind also der Startoffset, der Endoffset und der Offsetinkrement. Die ersten beiden geben dabei den Frequenzbereich an, in dem der Chirp arbeitet. Zur Berechnung wird folgende Formel[2] eingesetzt:

$$\text{offset} = \frac{f_0 \cdot (2^N - 1)}{f_{cl}}$$

Als Beispiel erhalten wir bei einer Frequenz $f_0 = 45kHz$ mit einem $N = 48$ Bit breiten Zähler und 33 MHz Systemtakt einen Offset von 383829513696. Diese hohe Bitbreite ist nötig, damit wir möglichst feine Abstufungen beim Anstieg der Frequenz erhalten. Der noch fehlende Parameter Offsetinkrement bestimmt die Steigung des Chirps und damit die Zuordnung von Entfernung zu Frequenz bei der Ausgabe. Wir haben uns für eine Erhöhung um 18092 entschieden, was einem Anstieg von $35kHz/s$ (also 5 Chirps pro Sekunde) entspricht. Dies hat zur Folge dass Töne von 100 Hz etwa 0.48 m und Töne von 1200 Hz etwa 5.8 m entsprechen. Mit

$$\text{offsetincr} = \frac{(\text{EndOffset} - \text{StartOffset}) \cdot \text{Chirps je Sekunde}}{f_{cl}}$$

wird der Offsetinkrement berechnet.

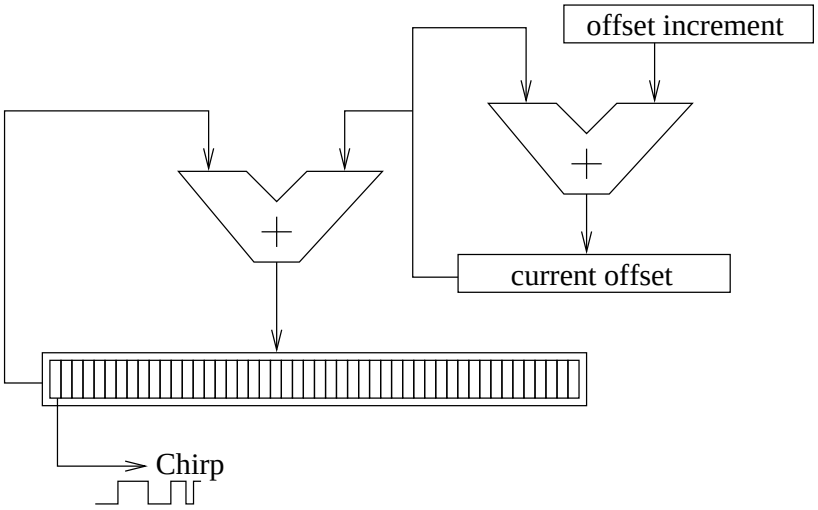


Abbildung 2.4: Aufbau des Chirpgenerators

2.3 Der Multiplizierer

Der Multiplizierer hat das empfangene Signal mit dem aktuellen Chirpsignal zu multiplizieren. Hierbei besteht das empfangene aus einer 8-Bit breiten vorzeichenbehafteten Zahl, während der Chirpgenerator nur ein Bit liefert, das als ± 1 interpretiert werden soll. Die Aufgabe reduziert sich dadurch im wesentlichen auf eine Negation, wenn das Chirpsample 0 ist bzw. der unveränderten Ausgabe des empfangenden Signals wenn das Chirpsample 1 ist. Zusätzlich wird noch ein Overflow-/Underflow-Signal ausgegeben. Dies entspricht im Regelfall direkt

[2] f_0 auszugebene Frequenz, f_{cl} Systemtakt, N Breite des Akkus

dem vom AD-Wandler kommenden Wert. Nur bei der Multiplikation von $-128 \cdot (-1)$ wird es direkt von dieser Komponente gesetzt, wobei das Ergebnis auf 127 abgerundet wird. Vom Rahmendesign wird das Überlauf-Bit auf eine LED geroutet, damit der Arbeitspunkt des AD-Wandlers grob überprüft werden kann.

2.4 Tiefpass-Filter

Nachdem das empfangene und das im Moment ausgeschickte Signal miteinander multipliziert sind, muss das Produkt von einem Tiefpass gefiltert werden (siehe Abbildung 1.7).

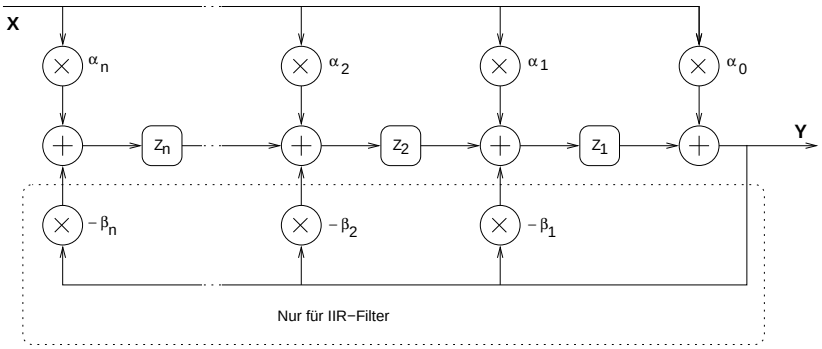


Abbildung 2.5: Schematischer Aufbau eines FIR-/IIR-Filters

Zur Realisierung digitaler Filter werden in der Regel FIR- oder IIR-Filter eingesetzt. Ein Blockschaltbild ist in Abbildung 2.5 gegeben. FIR-Filter sind rückkopplungsfrei, haben deswegen eine endlich lange Impulsantwort (FIR = Finite Impulse Response) und sind immer stabil. IIR-Filter sind rekursiv, was zu unendlich langen Impulsantworten (IIR = Infinite Impulse Response) und evtl. zu Stabilitätsproblemen führen kann. Für ein bestimmtes Dämpfungsverhalten sind IIR-Filter in der Regel jedoch kleinerer Ordnung als FIR-Filter. Da in unserem Ortungsgerät FPGAs eingesetzt werden, deren Speicherplatz eng bemessen ist, wurden in unserem Aufbau IIR-Filter verwendet.

Das Filter sollte ab 3 kHz anfangen zu dämpfen und bei Frequenzen ab 40 kHz eine Dämpfung von mindestens 54 db aufweisen (der Signal/Rausch-Abstand beträgt ca. $S = N \cdot 6db \approx 54db$ bei 9 Bit Auflösung (siehe [TieSch]).

Für unseren Aufbau geeignete Koeffizienten wurden von einem Projektteilnehmer mit Matlab ermittelt. Matlab lieferte uns ein zweistufiges Filter, zusammengesetzt aus zwei IIR-Filtern jeweils zweiter Ordnung. In Abbildung 2.6 sind die Koeffizienten angegeben. Abbildung 2.7 zeigt den Frequenzgang der beiden Stufen einzeln (gepunktet, gestrichelt) und auch den der Komposition (durchgezogen).

Unser Design verzichtet aus Platz- und Geschwindigkeitsgründen zugunsten von Festkommarechnung auf die Verwendung von Fliesskommaarithmetik. Werden alle Filter-Koeffizienten betragsmäßig kleiner 1 gewählt, so ist es dabei sogar möglich, auf die Repräsentation von Vorkommastellen aller Koeffizienten und aller Register zu verzichten. In der Praxis konnten nicht alle β-Koeffizienten betragsmäßig kleiner 1 gewählt werden. Daher war die Ergänzung einer Shift-Einheit notwendig (in Abbildung 2.8 dargestellt). Die neuen Koeffizienten erhält man aus den alten durch eine Skalierung mit dem Faktor $\frac{1}{2}$.

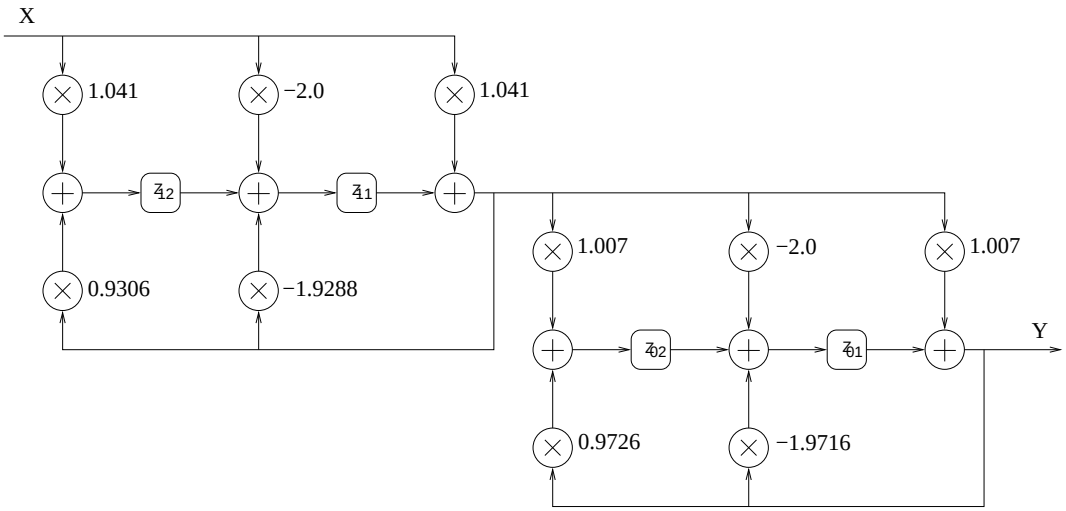


Abbildung 2.6: Das verwendete Filter: zwei gekoppelte IIR-Filter jeweils zweiter Ordnung

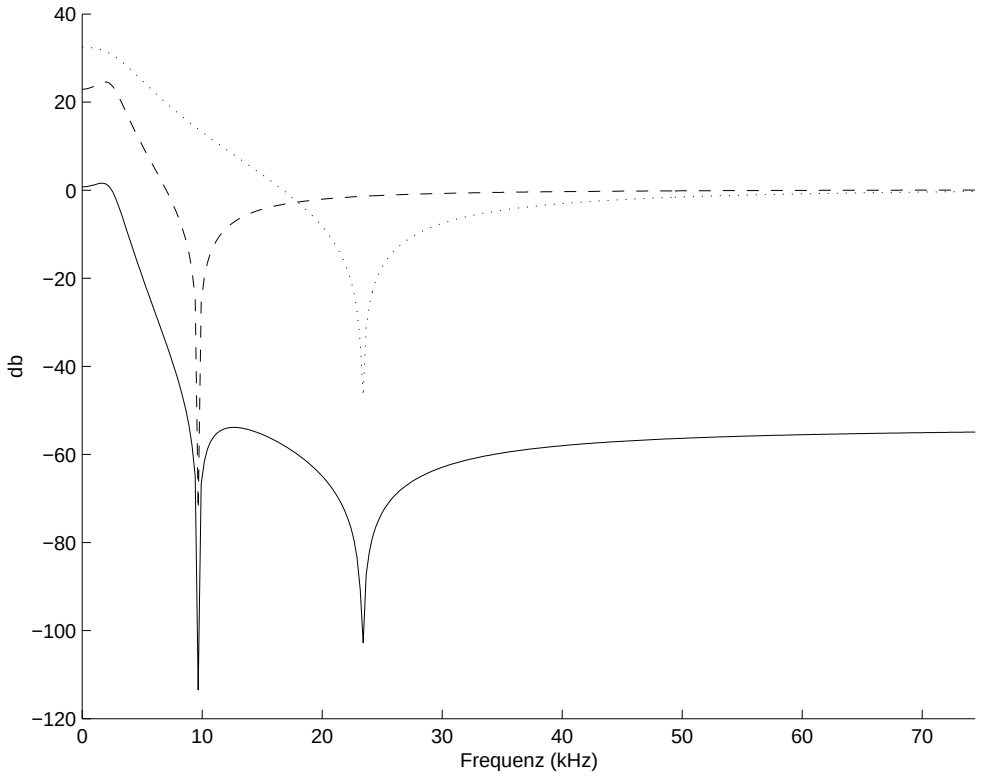


Abbildung 2.7: Der Frequenzgang des Filters aus Abbildung 2.6

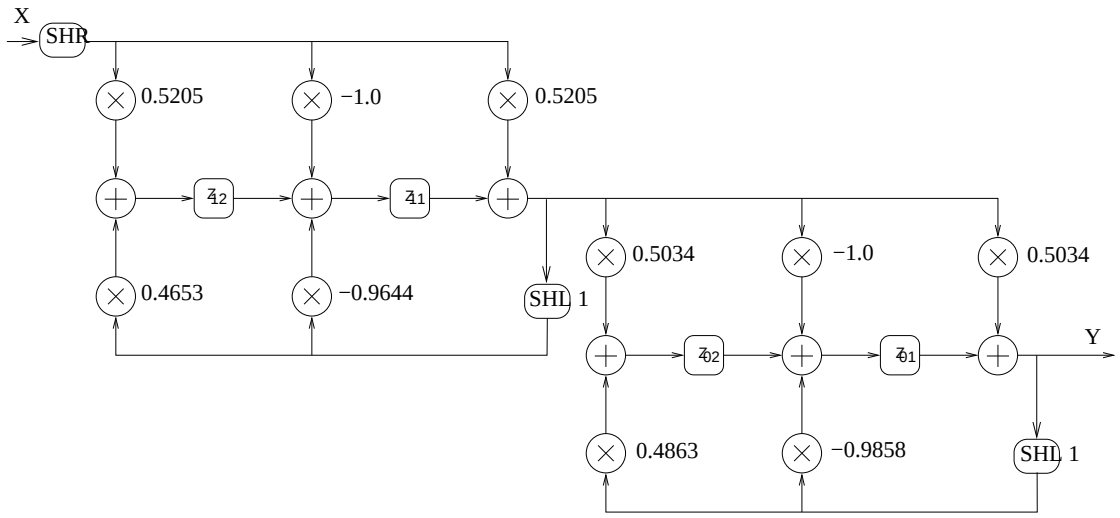


Abbildung 2.8: Filter mit optimierten Koeffizienten

Um ein Überlaufen der Akkumulatoren (entspricht der Aufnahme eines Wertes mit Betrag größer 1) zu verhindern, wird das Eingangssignal X durch eine Shiftung (nach rechts) in ein geeignetes Intervall skaliert. Wie weit skaliert wird, kann durch DIP-Switches eingestellt werden.

In Versuchen hat sich herausgestellt, dass Registerbreiten von 24 Bit und Koeffizientenbreiten von 12 Bit gute Ergebnisse liefern. Nicht mitgerechnet ist dabei die implizite 0 vor dem Komma.

Kapitel 3

Erweiterungen

3.1 Serialisierung des Filters

Der im letzten Kapitel beschriebene Ansatz zur Realisierung eines digitalen Filters hat, wenn es um seine Implementation in FPGAs geht, einen gravierenden Nachteil: seine Größe (gemessen in Logikzellen).

Wie in Abbildung 2.8 zu sehen, werden in einem naiven Ansatz zehn voneinander unabhängige Multiplikationseinheiten und sechs Additionseinheiten gleichzeitig verwendet. Besser wäre es, insgesamt nur einen Addierer und einen Multiplizierer bereitzustellen (evtl. zusammengefasst zu einer MAC[1]-Einheit), die dann sequentiell verwendet werden. Natürlich wird das Filter dadurch langsamer, werden doch mehrere Takte für die Verarbeitung eines Samples benötigt.

Um die Verwendung der MAC-Einheit zu koordinieren und um deren gleichzeitige doppelte Nutzung auszuschliessen, ist die zentrale Komponente der sequentiellen Variante des Filters ein Endlicher Automat. Sein Aufbau ist denkbar einfach (Abbildung 3.1): die zehn Zustaende $q_0 - q_9$ werden linear der Reihe nach erreicht. Jedesmal, wenn ein neues Sample am Eingang des Filters zu bearbeiten ist, wird der Automat in q_0 gestartet. Die nächsten neun Takte wechselt er jeweils von q_i nach q_{i+1}. Alle nachfolgenden Takte verharrt er in q_9, bis ein neues Sample zu bearbeiten ist und er in q_0 neu gestartet wird.

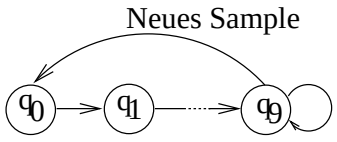


Abbildung 3.1: Endlicher Automat zur Kontrolle der MAC-Einheit

In welcher Transition welcher Teil des IIR-Filter gerechnet wird ist in Abbildung 3.2 und im Quellcode im Anhang zu sehen.

[1] Kurzform für Multiply+Accumulate

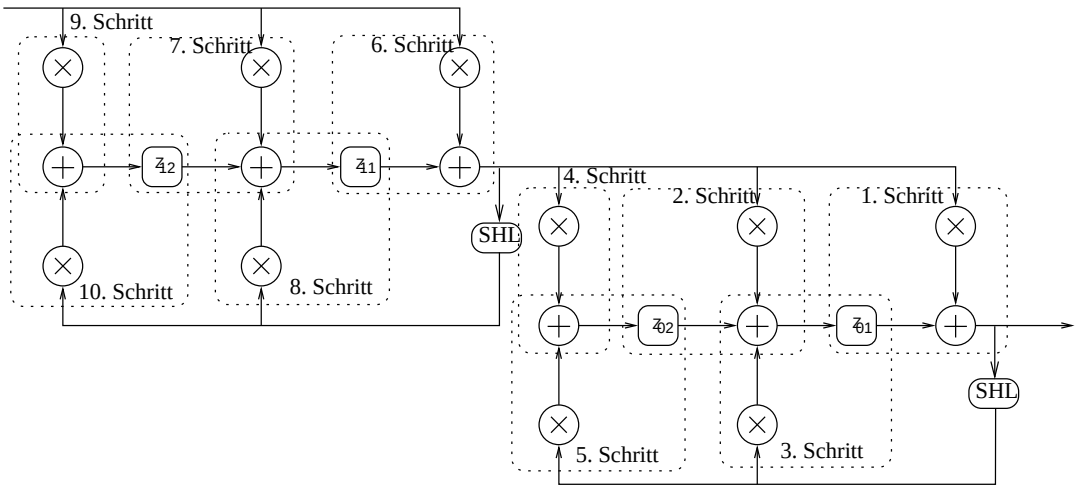


Abbildung 3.2: Kontrolle der MAC-Einheit durch einen Automaten

3.2 Fensterung

Die bisher beschriebene Implementation geht davon aus, dass wir eine konstante Steigung der Chirpfrequenz haben. Da dies leider technisch nicht machbar ist, haben wir regelmässig einen Frequenzsprung in dem Signal.

3.2.1 Idee

Dieser Sprung führt zu neuen Frequenzen und macht sich dadurch bemerkbar, das es keinen durchgehenden Ton gibt, sondern regelmässige Pausen/Störungen (etwa 5 mal pro Sekunde). Enstehen tun diese durch den Sprung der Chirpfrequenz von 52kHz zurück auf 45kHz. Als Folge entsteht im ausgegebenen Signal 5 mal pro Sekunde ein Frequenzsprung von beispielsweise 1kHz auf 6kHz für die Dauer von je $\frac{1}{35s}$ (siehe Kap. 1.3). Zu diesen Zeiten fehlt also das gewünschte Signal. Um dies zu beseitigen, werden zwei Chirps im Bereich von 45kHz bis 52kHz und im Bereich von 30kHz bis 37kHz mit Zeitversatz der halben Chirpperiode ausgesandt. Das empfangene Signal wird dann parallel in zwei Multiplizierern mit je einem der beiden Chirps multipliziert (Abb. 3.3). Danach werden die beiden Anteile gewichtet addiert. Bei der Wahl der Wichtung gibt es mehrere Möglichkeiten. Der Verlauf, wie in Abbildung 3.4 hat den Vorteil sehr einfach implementierbar zu sein. Jedoch ist der kritische Bereich am Chirpende nicht ausreichend ausgeblendet. Eine Wichtung nach Abbildung 3.5 hat den genannten Nachteil der unzureichenden Ausblendung nicht mehr. Eine Implementation so einer Kurve verlangt allerdings nach einer Tabelle bzw. einer recht aufwendigen Berechnung in Hardware. Sie ist daher für einen FPGA-Entwurf aufgrund der begrenzten Anzahl logischer Zellen nicht geeignet. Als Kompromiss wurde eine trapezförmige Gewichtung nach Abbildung 3.6 implementiert. Hier ist der Ausblendbereich ausreichend breit, um die unerwünschten Signale im Bereich des Chirpende auszublenden.

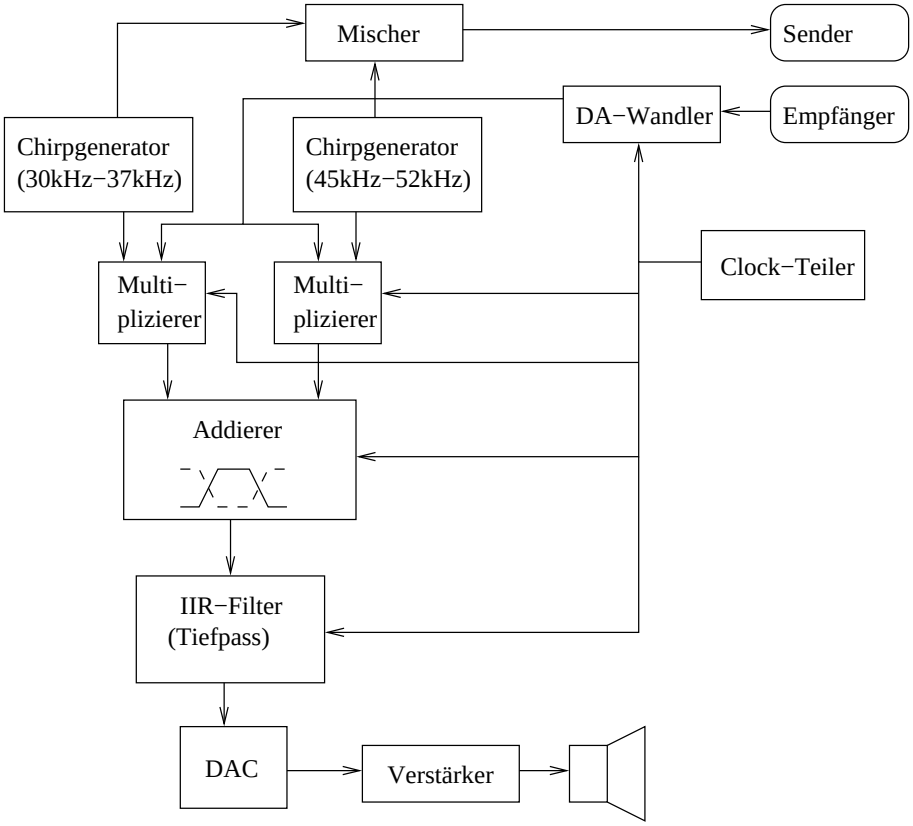


Abbildung 3.3: Struktur des FPGA-Entwurfs nach der Erweiterung mit der Fensterung

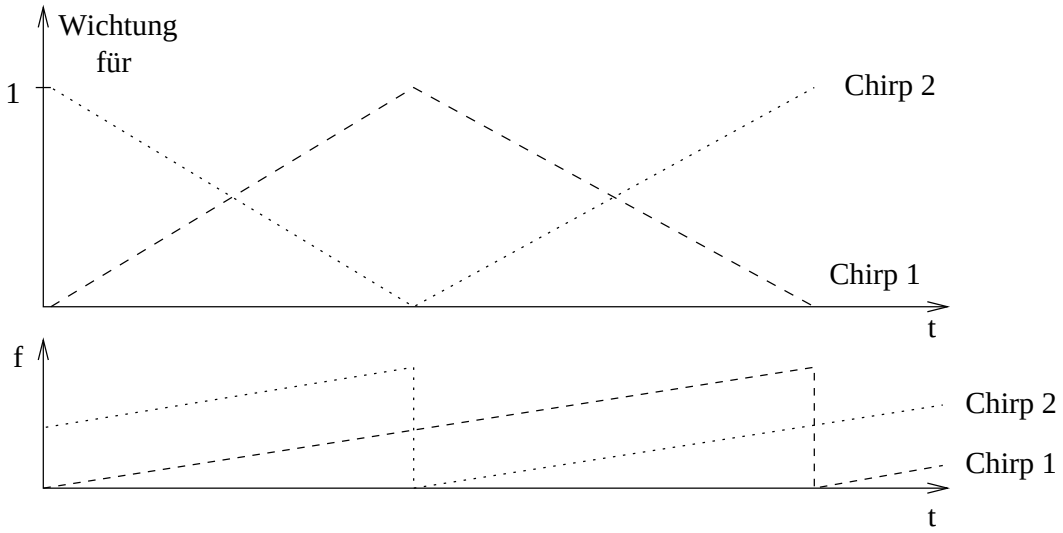


Abbildung 3.4: möglicher Verlauf des Fensters

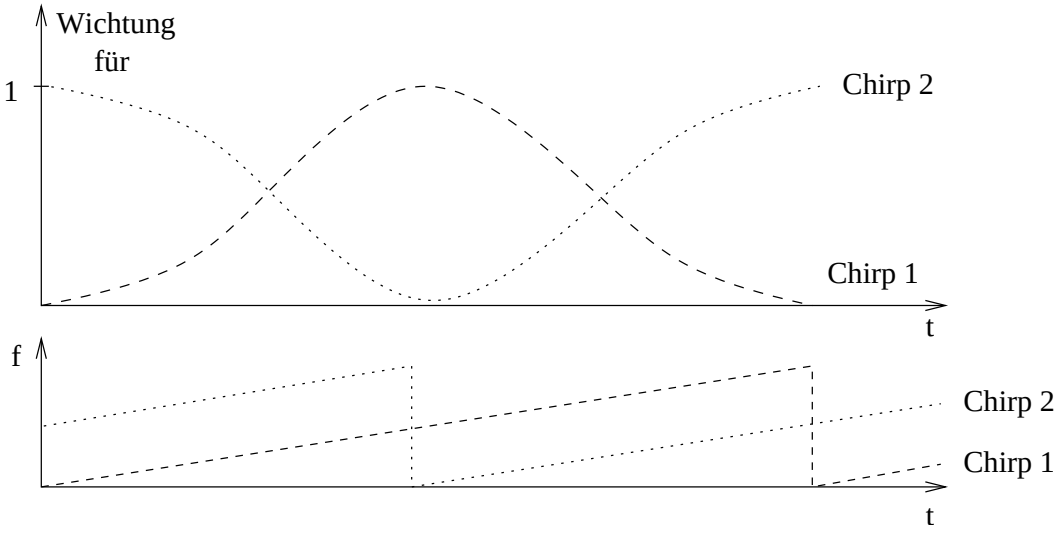


Abbildung 3.5: möglicher Verlauf des Fensters

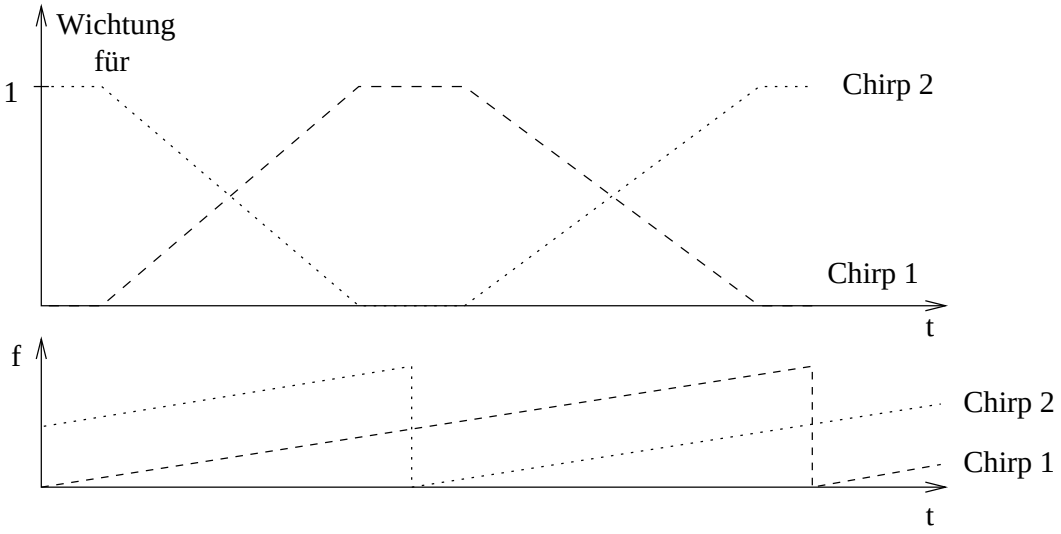


Abbildung 3.6: gewählter Trapezverlauf für das Fenster

3.2.2 Notwendige Erweiterungen der ursprünglichen Implementation

Gegenüber dem ersten Entwurf musste ein zweiter Chirpgenerator hinzugefügt werden, der um eine halbe Chirpperiode versetzt und in einem anderen Frequenzbereich arbeitet. Dies war durch anpassen einiger Konstanten erreicht. Dazu kommt noch ein zweiter, mit dem ersten identischer Multiplizierer. Der aufwendigste Teil dieser Erweiterung war der Addierer mit der Fenster-Gewichtung. Damit der Addierer synchron zum Chirp arbeitet, erhält dieser vom Chirpgenerator eine Position. Diese liegt in einem Bereich[2] von 0 für den Beginn bis 455526 für die komplette Chirpperiode. Dieser Bereich wurde nun in Segmente aufgeteilt in denen die Wichtung konstant (0 oder 1) ist, bzw. linear ansteigt oder abfällt (Abb. 3.6). Dabei wurden die Bereiche für Steigung und Gefälle so gewählt, dass ihre Breite in Samples eine zweier-Potenz ist, welche dann direkt als Wichtung in den Bereich 0 bis 1 gemappt werden kann.

[2] bestimmt durch den Offset-bereich im Chirpgenerator

Kapitel 4

Ausblick

Die Itensität des empfangenen Signals nimmt quadratisch mit dem zurückgelegten Weg ab. Als Folge sind die hohen Frequenzen im ausgegebenen Signal entsprechend schwächer als die tiefen Frequenzen. Dies könnte zwar durch ein entsprechendes Filter ausgeglichen werden, andererseits sind auf diese Weise nahe Objekte deutlicher und lauter zu hören, was der natürlichen Wahrnehmung entgegenkommt.

Wie aus der Abbildung 1.6 zu sehen ist, betreiben wir die Sender-/Empfänger-Kombination nicht im Resonanzbereich. Als Folge ist die Signalstärke nur etwa $\frac{1}{30}$ vom Maximum und vermindert dadurch die sinnvoll zu messenden Entfernungen. Für unserer Zwecke reicht die aber aus, um Gegenstände in für Räumen üblichen Entfernungen zu dedektieren, da der Nutzer nur einen Eindruck von seiner unmittelbaren Umgebung erhalten soll.

Ein weiteres Problem ergibt sich, wenn sich zwei Nutzer mit zwei verschiedenen Geräten gegenüber stehen. Es werden dann weiterere Frequenz zum Ausgabesignal hinzukommen, welche lauter als die eigentlich gewünschten Töne sind, weil die Ultraschallsignale nur den halben Weg zurücklegen müssen. Die zusätzlichen Frequenzen sind abhängig von dem Zeitversatz der beiden Chirps. Falls diese Differenz nicht konstant bleibt, z.B. weil die beiden Geräte mit einem minimal unterschiedlichen Systemtakt arbeiten, so wird diese Frequenz im Basisband bis $7kHz$ hin und her wandern. Es tritt eine Schwebung auf. Die Geschwindigkeit mit der dies geschieht ist um so größer, je weiter die beiden Systemtakte auseinanderliegen. Dies könnte man versuchen zu eleminieren, indem man noch weitere Frequenzbänder, zwischen denen man dynamisch umschaltet, nutzt. Dazu müsste das Ortungsgerät die empfangenden Signale daraufhin überprüfen, ob ein weiteres Gerät Signale aussendet, um dann einen anderen Bereich auszuwählen.

Anhang A

Quellcodes

A.1 nios_ wrap.vhd

```
-- -*-VHDL-*-
-- nios_wrap.vhd
--
-- NIOS-board wrapper   ajm     08-aug-2001
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
--use ieee.std_logic_arith.all;
use ieee.numeric_std.all;

entity nios_wrap is
port (  board_i_clk     : in  std_logic;        -- 33MHz  clock
        board_i_clrN    : in  std_logic;        -- board reset => don't use!!!
                                                --       init APEX from flash
        ------------------------------------------------------------------
        -- prototype connector 5.0V             -- connected with level-shift
        ---------------------------------------- JP11, JP12, JP13
        prot5_o_selN    : out std_logic;        -- '0' enables I/O
        prot5_b_io      : inout std_logic_vector(39 downto 0);
        prot5_i_clk     : in  std_logic;        --
        prot5_o_vee     : out std_logic;        --
--      ------------------------------------------------------------------
--      -- prototype connector 3.3V             -- direct connection
--      ---------------------------------------- JP8, JP9, JP10
--      prot3_o_selN    : out std_logic;        --
--      prot3_b_io      : inout std_logic_vector(39 downto 0);
--      prot3_i_clk     : in  std_logic;        --
--      ------------------------------------------------------------------
--      -- SDRAM connector                      -- direct connection
--      ---------------------------------------- J2
--      sdram_o_addr    : out std_logic_vector(13 downto 0);
```

```
--      sdram_b_data    : inout std_logic_vector(63 downto 0);
--      sdram_o_ba      : out std_logic_vector(1 downto 0);
--      sdram_o_dqm     : out std_logic_vector(7 downto 0);
--      sdram_o_casN    : out std_logic;        --
--      sdram_o_rasN    : out std_logic;        --
--      sdram_o_weN     : out std_logic;        --
--      sdram_o_s0N     : out std_logic;        --
--      sdram_o_s1N     : out std_logic;        --
--      sdram_o_dnu     : out std_logic;        --
--      sdram_o_sda     : out std_logic;        --
--      sdram_o_scl     : out std_logic;        --
--      sdram_o_cke0    : out std_logic;        --
--      sdram_o_cke1    : out std_logic;        --
--      ------------------------------------------------------------------------
--      -- PCI mezzanine connectors             -- direct connection
--      ----------------------------------------- JN1, JN2
--      pcm_b_io        : inout std_logic_vector(75 downto 0);
        ------------------------------------------------------------------------
        -- user I/O                             -- '0' is default-output!!!
        ------------------------------------------------------------------------
        user_i_but      : in  std_logic_vector(3 downto 0);  -- '0'     pressed
        user_i_sw       : in  std_logic_vector(7 downto 0);  -- '0'/'1' switch
        user_o_led0     : out std_logic;        -- '0' off
        user_o_led1     : out std_logic;        -- '0' off
        user_o_hex0a    : out std_logic;        -- '1' off
        user_o_hex0b    : out std_logic;        -- '1' off
        user_o_hex0c    : out std_logic;        -- '1' off
        user_o_hex0d    : out std_logic;        -- '1' off
        user_o_hex0e    : out std_logic;        -- '1' off
        user_o_hex0f    : out std_logic;        -- '1' off
        user_o_hex0g    : out std_logic;        -- '1' off
        user_o_hex0dp   : out std_logic;        -- '1' off
        user_o_hex1a    : out std_logic;        -- '1' off
        user_o_hex1b    : out std_logic;        -- '1' off
        user_o_hex1c    : out std_logic;        -- '1' off
        user_o_hex1d    : out std_logic;        -- '1' off
        user_o_hex1e    : out std_logic;        -- '1' off
        user_o_hex1f    : out std_logic;        -- '1' off
        user_o_hex1g    : out std_logic;        -- '1' off
        user_o_hex1dp   : out std_logic;        -- '1' off
--      ------------------------------------------------------------------------
--      -- serial port                          -- direct connection
--      ----------------------------------------
--      ser_o_cts       : out std_logic;
--      ser_o_txd       : out std_logic;
--      ser_i_rxd       : in  std_logic;
--      ser_i_rts       : in  std_logic;
```

```
        ------------------------------------------------------------------
        -- SRAM and FLASH                       -- chip select only
        ------------------------------------------
        sram0_o_csN     : out std_logic;        -- '1' to disable
        sram1_o_csN     : out std_logic;        -- '1' to disable
        flash_o_csN     : out std_logic);       -- '1' to disable
end entity nios_wrap;

--------------------------------------------------------------------------
architecture tb of nios_wrap is
signal nreset : std_logic;
signal sample_clk : std_logic;
signal board_clk : std_logic;
signal chirp_data1 : std_logic;
signal chirp_data2 : std_logic;
signal Sample_chirp_data1 : std_logic;
signal Sample_chirp_data2 : std_logic;
signal AD_Dout : signed(7 downto 0);
signal AD_Overflow : std_logic;
signal Mul_Dout1 : signed(7 downto 0);
signal Mul_Dout2 : signed(7 downto 0);
signal Mul_Overflow1 : std_logic;
signal Mul_Overflow2 : std_logic;
signal Filter_Overflow : std_logic;
signal FilterOut : signed (23 downto 0);
signal unsigned_Filter_out : unsigned (7 downto 0);
signal Shift_Out : signed ( 23 downto 0 );
signal shift_val : unsigned(3 downto 0);
signal chirpedge1:  std_logic;
signal chirpedge2:  std_logic;
signal preshiftOut: unsigned (3 downto 0);
signal bcd : unsigned(7 downto 0);
signal last_chirp_edge : std_logic;
signal last_chirp_edge2 : std_logic;
signal chirp_pos1 : unsigned ( 18 downto 0 );
signal chirp_pos2 : unsigned ( 18 downto 0 );
signal henning_Dout : signed (23 downto 0);
signal filter_in : signed (23 downto 0);
signal preshift : unsigned (3 downto 0);

component chirp_gen1 is
        port ( clk : in std_logic;
               reset : in std_logic;
               data : out std_logic;
               chirp_pos : out unsigned (18 downto 0);
               chirpedge : out std_logic );
end component;
```

```
component chirp_gen2 is
        port ( clk : in std_logic;
               reset : in std_logic;
               data : out std_logic;
               chirp_pos : out unsigned (18 downto 0);
               chirpedge : out std_logic );
end component;

component shifter
  port (
    Din       : in  signed ( 7 downto 0 );
    reset     : in std_logic;
    clk       : in std_logic;
    shift_val : in  unsigned ( 3 downto 0 );
    Dout      : out signed ( 23 downto 0 ));
end component;

component filter
    port ( X : in signed  (24-1  downto 0);
           Y : out signed (24-1  downto 0);
           reset: in STD_LOGIC;
           clk: in STD_LOGIC;
           overflow: out std_logic;
           chirpedge: in std_logic;
           preshiftIn: in unsigned (3 downto 0);
           preshiftOut: out unsigned (3 downto 0));
end component;

component clk_div
   generic (
     clock_divisor : integer := 6 );
   port ( clk_in : in std_logic;
          reset : in std_logic;
          clk_out : out std_logic );
end component;

component mul is
   port ( Din : in signed ( 7 downto 0 );
          OverflowIn : in std_logic;
          chirpIn : in std_logic;
          Dout : out signed ( 7 downto 0 );
          OverflowOut : out std_logic;
          reset : in std_logic );

end component;
```

```
component hanningadd is
   port (
     chirp_pos     : in unsigned (18 downto 0);
     clk           : in std_logic;
     dIn1          : in signed (7 downto 0);
     dIn2          : in signed (7 downto 0);
     dOut          : out signed (23 downto 0));
end component;

  function bcd27seg (a: unsigned(3 downto 0)) return unsigned is
    variable res : unsigned(7 downto 0);
  begin
    case a is
      when "0000" => res := "01111110";
      when "0001" => res := "00110000";
      when "0010" => res := "01101101";
      when "0011" => res := "01111001";
      when "0100" => res := "00110011";
      when "0101" => res := "01011011";
      when "0110" => res := "01011111";
      when "0111" => res := "01110000";
      when "1000" => res := "01111111";
      when "1001" => res := "01111011";
      when "1010" => res := "01110111";
      when "1011" => res := "00011111";
      when "1100" => res := "01001110";
      when "1101" => res := "00011101";
      when "1110" => res := "01001111";
      when "1111" => res := "01000111";
      when others => res := "00110110";
    end case;
    return res;
  end bcd27seg;

begin
  sram0_o_csN  <= '1';                        -- disable external SRAM
  sram1_o_csN  <= '1';                        --    -
  flash_o_csN  <= '1';                        --    -           FLASH

  prot5_o_selN <= '0';                        -- enable 5V I/O

  nreset <= user_i_but(3);


  board_clk <= board_i_clk;

  --    45kHz - 52kHz
```

```
chirp1 : chirp_gen1  port map ( board_clk, nreset, chirp_data1, chirp_pos1, chirpedge1 );

--    45kHz - 52kHz (beginnt bei ca 48.5kHz)
chirp2 : chirp_gen2  port map ( board_clk, nreset, chirp_data2, chirp_pos2, chirpedge2);


clock_div : clk_div port map (board_clk, nreset, sample_clk);

-- purpose: Daten vom AD-Wandler holen
-- type   : combinational
-- inputs : sample_clk, reset
-- outputs: AD_Dout, AD_Overflow
AD: process (sample_clk, nreset, chirp_data1)
begin  -- process AD
  if ( nreset = '0' ) then
    AD_Dout <= (others => '0');
    AD_Overflow <= '0';
  elsif rising_edge(sample_clk) then  -- sample on falling clk
    AD_Dout <= signed(prot5_b_io(37 downto 30));
    AD_Overflow <= prot5_b_io(38);
    Sample_chirp_data1 <= chirp_data1;
    Sample_chirp_data2 <= chirp_data2;
  end if;
end process AD;

multi1 : mul port map (AD_Dout, AD_Overflow, Sample_chirp_data1, Mul_Dout1, Mul_Overflow1, nreset);
multi2 : mul port map (AD_Dout, AD_Overflow, Sample_chirp_data2, Mul_Dout2, Mul_Overflow2, nreset);

henningadder : hanningadd port map (chirp_pos1, sample_clk, Mul_Dout1, Mul_Dout2,henning_Dout);

shift_val <= unsigned(user_i_sw(3 downto 0));
shiftercom : shifter port map (Mul_Dout1, nreset, sample_clk, shift_val, Shift_Out);

user_o_hex0a <= not shift_val(0);
user_o_hex0b <= not shift_val(1);
user_o_hex0c <= not shift_val(2);
user_o_hex0d <= not shift_val(3);



preshift(3) <= user_i_sw(3);
preshift(2) <= user_i_sw(2);
preshift(1) <= user_i_sw(1);
preshift(0) <= user_i_sw(0);

-- filter-reset ist low-active
filtercom : filter port map (filter_in, FilterOut, nreset,sample_clk, Filter_Overflow,
                             chirpedge1,preshift, preshiftOut);
```

```
-- Index anpassen
unsigned_Filter_out <=  to_unsigned(to_integer(FilterOut(23 downto 16)) + 128, 8);
-- std_logic_vector(FilterOut(23 downto 16));
prot5_b_io(7 downto 0) <= std_logic_vector(unsigned_Filter_out);

seg: process (preshiftOut)
  variable res : unsigned (7 downto 0);
begin  -- process 7seg
  res := bcd27seg(preshiftOut);
  user_o_hex1a <= not  res(6);
 user_o_hex1b <= not  res(5);
 user_o_hex1c <= not  res(4);
 user_o_hex1d <= not  res(3);
 user_o_hex1e <=  not res(2);
 user_o_hex1f <=  not res(1);
 user_o_hex1g <=  not res(0);
 user_o_hex1dp <=  not res(7);
end process seg;


disableChirp2: process (user_i_but, chirp_data1, chirp_data2)
begin  -- process disableChirp2
  if user_i_but(2)='1' then
    prot5_b_io(27) <= chirp_data1;
    prot5_b_io(28) <= chirp_data2;
    filter_in <= henning_Dout;
  else
    prot5_b_io(27) <= chirp_data1;
    prot5_b_io(28) <= '0';
    filter_in <= shift_Out;
  end if;
end process disableChirp2;

prot5_b_io(39) <= sample_clk;

process (Mul_Overflow1, Mul_Overflow2, nreset, sample_clk, chirpedge1)
begin  -- process
  if ( nreset = '0' ) then
    user_o_led0 <= '0';
    last_chirp_edge <= chirpedge1;
  elsif rising_edge(sample_clk) then
      if Mul_Overflow1='1' or Mul_Overflow2='1' then
            user_o_led0 <=  '1';
      elsif  chirpedge1 /= last_chirp_edge then
        last_chirp_edge <= chirpedge1;
        user_o_led0 <= '0';
```

```
        end if;
    end if;
  end process;

  process (Filter_Overflow, nreset, sample_clk, chirpedge1)
  begin  -- process
    if ( nreset = '0' ) then
      user_o_led1 <= '0';
      last_chirp_edge2 <= chirpedge1;
    elsif rising_edge(sample_clk) then
        if Filter_Overflow='1' then
              user_o_led1 <=  '1';
        elsif  chirpedge1 /= last_chirp_edge2 then
          last_chirp_edge2 <= chirpedge1;
          user_o_led1 <= '0';
        end if;
    end if;
  end process;

end architecture tb;
```

A.2 clk-div.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.ALL;
--use ieee.std_logic_arith.ALL;

entity clk_div is
   generic (
     clock_divisor : integer := 8 );
   port ( clk_in : in std_logic;
          reset : in std_logic;
          clk_out : out std_logic );
end clk_div;

architecture my_clk_div of clk_div is
  signal akku : unsigned ( clock_divisor - 1 downto 0);
  signal  tmp_clk_out : std_logic;
begin  -- chirp_gen

  clk_out <= tmp_clk_out;
  tmp_clk_out <= akku(akku'length -1);

  incr_akku: process (clk_in, reset)
```

```
  begin  -- process incr_akku
    if reset='0' then
      akku <= (others => '0');
    elsif rising_edge(clk_in) then
      akku <= akku + 1;
    end if;
  end process incr_akku;


end my_clk_div;
```

A.3 chirp-gen1.vhd

```
-- -*-VHDL-*-
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity  chirp_gen1 is
        port ( clk : in std_logic;
               reset : in std_logic;
               data : out std_logic;
               chirp_pos     : out unsigned (18 downto 0);  -- index fuer Hanning-Tab
               chirpedge : out std_logic );
end chirp_gen1;

architecture my_chirp_gen1 of chirp_gen1 is
  signal akku : unsigned (47 downto 0);
  signal new_akku : unsigned (47 downto 0);
  signal cur_offset : unsigned (47 downto 0);
  signal tmp_pos : unsigned (47 downto 0);
  signal out_data : std_logic;
  signal offset_inc_time : unsigned (3 downto 0);
  signal local_chirp_edge : std_logic;                        --  11011110011011001101101010110011101 0
  constant start_offset : unsigned (47 downto 0) := "000000000101100101011110000001011001010111100000";
  constant incr_offset : unsigned (47 downto 0)  := "000000000000000000000000000000000010001101010110";
  constant  end_offset : unsigned (47 downto 0)  := "000000000110011101000100110100110100000100011010" ;

begin  -- chirp_gen

  out_data <= akku(akku'length - 1);
  data <= out_data;
  chirpedge <= local_chirp_edge;

  chirp_pos <= tmp_pos(tmp_pos'length - 13 downto tmp_pos'length - chirp_pos'length - 12  );
```

```
-- purpose: berechnung der cirp position
-- type   : combinational
-- inputs : clk, reset
-- outputs:
calcChirpPos: process (clk, reset, cur_offset)
begin  -- process calcChirpPos
  if reset = '0' then
    tmp_pos <= cur_offset;
  elsif rising_edge(clk) then
    tmp_pos <= cur_offset - start_offset;  -- chirp_pos geht nur bis 1101111001101100110
  end if;
end process calcChirpPos;

-- purpose: Offset inkrementieren
-- type   : combinational
-- inputs : clk
-- outputs:
offset_inc: process (clk, reset, local_chirp_edge)
begin  -- process offset_inc
  if (reset = '0' ) then
    cur_offset <= start_offset;
    local_chirp_edge <= '0';
    offset_inc_time <= (others => '0');
  elsif (rising_edge(clk) ) then
      if (cur_offset > end_offset) then
        local_chirp_edge <= not local_chirp_edge;
        cur_offset <= start_offset;
      else
        cur_offset <= cur_offset + incr_offset ;
      end if;
  end if;
end process offset_inc;

new_akku <= akku+ cur_offset;

-- purpose: akku inkrementieren
-- type   : combinational
-- inputs : clk
-- outputs:
incr_akku: process (clk,reset,out_data)
begin  -- process incr_akku
  if (reset = '0' ) then
    akku <= (others => '0');
  elsif( rising_edge(clk) ) then
    if( new_akku(new_akku'length - 1) = '0' and out_data='1' ) then
      akku <= (others => '0');
```

```
      elsif( new_akku(new_akku'length - 1) = '1' and out_data='0' ) then
        akku <= ( 47 => '1' , others => '0');
      else
        akku <= new_akku;
      end if;
  end if;
  end process incr_akku;

end my_chirp_gen1;
```

A.4 chirp-gen2.vhd

```
-- -*-VHDL-*-
library ieee;
use ieee.std_logic_1164.all;
--use ieee.std_logic_arith.all;
use ieee.numeric_std.all;

entity  chirp_gen2 is
        port ( clk : in std_logic;
               reset : in std_logic;
               data : out std_logic;
                chirp_pos     : out unsigned (18 downto 0);  -- index fuer Hanning-Tab
                chirpedge : out std_logic );
end chirp_gen2;

architecture my_chirp_gen2 of chirp_gen2 is
  signal akku : unsigned (47 downto 0);
  signal new_akku : unsigned (47 downto 0);
  signal cur_offset : unsigned (47 downto 0);
  signal out_data : std_logic;
  signal offset_inc_time : unsigned (3 downto 0);
  signal local_chirp_edge : std_logic;
  -- chirp von 30 bis 37 kHz mit 35kHz/s
  constant start_offset : unsigned (47 downto 0) := "000000000011101110010100000000111011100101000000";
  constant incr_offset : unsigned (47 downto 0)  := "000000000000000000000000000000000010001101010110";
  constant  end_offset : unsigned (47 downto 0)  := "000000000100100101111010110100010110010001111010";

begin  -- chirp_gen

  out_data <= akku(akku'length - 1);
  data <= out_data;
  chirpedge <= local_chirp_edge;

  chirp_pos <= akku(akku'length - 1 downto akku'length - chirp_pos'length);
```

```
  -- purpose: Offset inkrementieren
  -- type   : combinational
  -- inputs : clk
  -- outputs:
  offset_inc: process (clk, reset, local_chirp_edge)
  begin  -- process offset_inc
    if (reset = '0' ) then
      cur_offset <= "00000000010000101000011101101010100011101101l101";
      local_chirp_edge <= '0';
      offset_inc_time <= (others => '0');
    elsif (rising_edge(clk) ) then
        if (cur_offset > end_offset) then
          local_chirp_edge <= not local_chirp_edge;
          cur_offset <= start_offset;
        else
          cur_offset <= cur_offset + incr_offset ;
        end if;
    end if;
  end process offset_inc;

  new_akku <= akku+ cur_offset;

  -- purpose: akku inkrementieren
  -- type   : combinational
  -- inputs : clk
  -- outputs:
  incr_akku: process (clk,reset,out_data)
  begin  -- process incr_akku
    if (reset = '0' ) then
      akku <= (others => '0');
    elsif( rising_edge(clk) ) then
      if( new_akku(new_akku'length - 1) = '0' and out_data='1' ) then
        akku <= (others => '0');
      elsif( new_akku(new_akku'length - 1) = '1' and out_data='0' ) then
        akku <= ( 47 => '1' , others => '0');
      else
        akku <= new_akku;
      end if;
  end if;
  end process incr_akku;

end my_chirp_gen2;
```

A.5 stufe_template.vhd

```
-- -*-VHDL-*-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use ieee.std_logic_arith.ALL;
use IEEE.numeric_std.ALL;

entity iir_filter**nummer** is
        port ( Xpre : in signed  (**regbreite** - 1  downto 0);
               Y : out signed (**regbreite** - 1  downto 0);
               reset: in STD_LOGIC;
               clk: in STD_LOGIC;
               overflow_fpm: out STD_LOGIC);
end iir_filter**nummer**;

architecture arch_filter of iir_filter**nummer** is
-------------------------------------------------------------------------
  constant regbreite   : integer := **regbreite**;
  constant order       : integer := **order**;     -- die ordnung des filters
  constant koeffbreite : integer := **koeffbreite**;
  constant beta0       : integer := **beta0**;     -- der rightshift bei beta(0)
  constant preshiftr   : integer := **preshiftr**;  -- der preshift am eingang
  -- typ fuer koeffizienten
  type Tkoeff is array (0 to order) of signed(koeffbreite - 1 downto 0);
  constant alpha : Tkoeff := (**alphakoeffizienten**);
  constant beta : Tkoeff := (**betakoeffizienten**);
  ---------------------------------------------------------------------------

  constant maxpos      : signed(koeffbreite -1 downto 0):= ('0',others => '1') ;  --
  -- groesste positive zahl (m.a.W.: die 1)

  signal y_scal : signed(regbreite - 1 downto 0);
  signal X : signed(regbreite - 1  downto 0);
  signal overflow_reg : std_logic_vector(3 * order + 2 downto 0);

  type Tregarr is array (1 to order) of signed(regbreite - 1 downto 0);  -- speichertyp fuer zustaende
  signal z : Tregarr;                -- die zustaende

**real**  function vec2real (arg : signed ) return real is
**real**    variable ret  : real := 0.0;
**real**    variable x    : real := 0.5;
**real**  begin
**real**    assert arg'left > arg'right
**real**      severity error;
**real**    if (arg(arg'left) = '1')  then ret := -1.0;
**real**    end if;
```

```
**real**    for i in arg'left-1 downto arg'right loop
**real**      if (arg(i) = '1')       then ret := ret + x;
**real**      end if;
**real**      x := x/2.0;
**real**    end loop;
**real**    return ret;
**real**  end function vec2real;

  -- purpose: fixpointmul
  function fixpointmul (a: signed(koeffbreite - 1 downto 0);
                        b: signed(regbreite - 1 downto 0)
                       ) return signed is
    variable prod : signed(koeffbreite + regbreite - 1 downto 0);
    variable ovflw: std_logic;
  begin
    if (a = maxpos) then -- max pos Zahl soll als 1 interpretiert werden
      return '0' & b;
    else
      prod := (a * b);
      ovflw := prod(prod'left - 1) XOR prod(prod'left);
      return ovflw
           & prod(prod'left)
           & prod((prod'left - 2) downto (prod'left - 2 - regbreite + 2));
    end if;
  end fixpointmul;

  function fixpointadd(a: signed(regbreite - 1 downto 0);
                       b: signed(regbreite - 1 downto 0);
                       c: signed(regbreite - 1 downto 0)) return signed is
    variable zw : signed(regbreite downto 0);
    variable zwovf: std_logic;
  begin
    zw := (a(a'left) & a) + (b(b'left) & b);
    zwovf := zw(zw'left) xor zw(zw'left - 1);
    zw := zw + (c(c'left) & c);
    zwovf := zwovf or (zw(zw'left) xor zw(zw'left - 1));
    return zwovf & zw(zw'left - 1 downto 0);  -- links ist ovf-bit
  end fixpointadd;

  function fixpointadd(a: signed(regbreite - 1 downto 0);
                       b: signed(regbreite - 1 downto 0)) return signed is
    variable zw : signed(regbreite downto 0);
    variable zwovf: std_logic;
  begin
    zw := (a(a'left) & a) + (b(b'left) & b);
    zwovf := zw(zw'left) xor zw(zw'left - 1);
    return zwovf & zw(zw'left - 1 downto 0);  -- links ist ovf-bit
```

```
  end fixpointadd;


**real**  signal realtmpy : real;
**real** type Trealz is array (1 to order) of real;
**real**  signal realz : Trealz;
begin  -- filter

  x <= shift_right(Xpre,preshiftr);

  -- purpose: Letzte Filterstufe
  LetzteStufe: process (z, x)
    variable tmp_y : signed(regbreite - 1 downto 0);
    variable prodoben : signed(regbreite downto 0); -- overflow und rechenerg
    variable summe : signed(regbreite downto 0);
    variable idx : integer;
    variable shiftovf: std_logic;
  begin  -- process LetzteStufe
    shiftovf := '0';
    prodoben := fixpointmul(alpha(0), x);
    overflow_reg(3 * order) <= prodoben(prodoben'left);
    summe := fixpointadd(prodoben(prodoben'left-1 downto 0), z(1));
    tmp_y := summe(summe'left -1 downto 0);
    overflow_reg(3 * order + 2) <= summe(summe'left);
    y <= tmp_y;
    y_scal <= shift_left(tmp_y, beta0);
    for idx in beta0  downto 1 loop
      if tmp_y(tmp_y'left) /= tmp_y(tmp_y'left - idx) then
        shiftovf := '1';
      end if;
    end loop;  -- idx
    overflow_reg(3 * order + 1) <= shiftovf;
**real**    realtmpy <= vec2real(tmp_y);
  end process LetzteStufe;

  -- purpose: erste stufe
  ErsteStufe: process(clk, reset)
    variable prodoben : signed(regbreite downto 0);
    variable produnten : signed(regbreite downto 0);
    variable summe : signed(regbreite downto 0);
  begin  -- process ErsteStufe
    if reset = '0' then
      z(order) <= (others => '0');
    elsif rising_edge(clk) then
      prodoben := fixpointmul(alpha(order), x);
      produnten := fixpointmul(beta(order), y_scal);
      overflow_reg(0) <= prodoben(prodoben'left);
```

```
      overflow_reg(1) <= produnten(produnten'left);
      summe := fixpointadd(prodoben(prodoben'left-1 downto 0),
                        - produnten(produnten'left-1 downto 0));
      z(order) <= summe(summe'left - 1 downto 0);
      overflow_reg(2) <= summe(summe'left);

**real**      realz(order) <= vec2real(prodoben(prodoben'left-1 downto 0)
**real**                               - produnten(produnten'left-1 downto 0));
    end if;
  end process ErsteStufe;

  -- purpose: die mittleren stufen
**ordergt2**  genlabel: for n in 1 to order - 1 generate -- schlechte software ist schlecht!
    MittlereStufe: process (clk, reset)
**ordereq2**  constant n: integer := 1;
      variable prodoben : signed(regbreite downto 0);
      variable produnten : signed(regbreite downto 0);
      variable summe : signed(regbreite downto 0);
    begin
      if reset = '0' then
        z(n) <= (others => '0');
      elsif rising_edge(clk) then
        prodoben := fixpointmul(alpha(n), x);
        produnten := fixpointmul(beta(n), y_scal);
        overflow_reg(3 * n) <= prodoben(prodoben'left);
        overflow_reg(3 * n + 1) <= produnten(produnten'left);

        summe := fixpointadd(prodoben(prodoben'left-1 downto 0),
                        - produnten(produnten'left-1 downto 0),
                            z(n + 1));
      z(n) <= summe(summe'left - 1 downto 0);
      overflow_reg(3 * n + 2) <= summe(summe'left);

**real**      realz(n) <= vec2real(z(n + 1) + prodoben(prodoben'left-1 downto 0)
**real**                             - produnten(produnten'left-1 downto 0));
      end if;
    end process MittlereStufe;
**ordergt2**  end generate genlabel;

  process (overflow_reg)   -- falls in *einer* multiplikation ein overflow
  begin                    -- stattfand, soll overflow_fpm gesetzt werden
    overflow_fpm <= '0';
    for n in overflow_reg'range loop
      if overflow_reg(n) = '1' then
        overflow_fpm <= '1';
      end if;
    end loop;
```

```
  end process;

end arch_filter;
```

A.6 mul.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.ALL;
--use ieee.std_logic_arith.ALL;

entity mul is
   port ( Din : in signed ( 7 downto 0 );
          OverflowIn : in std_logic;
          chirpIn : in std_logic;
          Dout : out signed ( 7 downto 0 );
          OverflowOut : out std_logic;
          reset : in std_logic );

end mul;

architecture my_mul of mul is
signal tmp_d : signed (7 downto 0);
begin  -- my_mul

 -- purpose: Multiplikation des empfangenen mit dem ausgesandten Signal
 -- type   : combinational
 -- inputs : reset, Din, OverflowIn, chirpIn
 -- outputs: Dou, OverflowOut
 multiply: process (reset, Din, OverflowIn, chirpIn)
 begin  -- process multiply
   if reset='0' then
     Dout <= (others => '0');
     OverflowOut <= '0';
   elsif chirpIn='1' then
     OverflowOut <= OverflowIn;
     Dout <= Din;
   else
     if OverflowIn='1' then
       OverflowOut <= '1';
       Dout <= signed( not std_logic_vector(Din) );
     elsif Din = (-128) then
```

```
      Dout <= ( 7=>'0', others => '1');
      OverflowOut <= '1';
    else
      Dout <= -Din; -- signed( not std_logic_vector(tmp_d) );
      OverflowOut <= '0';
    end if;
  end if;
end process multiply;

end my_mul;
```

A.7 shifter.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.ALL;
--use ieee.std_logic_arith.ALL;

entity shifter is
  -- constant regbreite : integer := 24;

   port ( Din : in signed ( 7 downto 0 );
          reset : in std_logic;
          clk   : in std_logic;
          shift_val : in unsigned (3 downto 0);
          Dout : out signed ( 23 downto 0 ));

end shifter;

architecture my_shifter of shifter is
signal tmp_d : signed (Dout'length -1  downto 0);

begin  -- my_shifter

  tmp_d(Dout'length - 1 downto Dout'length - 8) <= Din(7 downto 0);
  tmp_d(Dout'length - 9 downto 0) <= (others => '0');

  pr: process (clk, reset)
  begin  -- process pr
    if (reset = '0') then
      Dout <= (others => '0');
    else
      if( rising_edge(clk) ) then
       -- Dout <= shift_right(tmp_d, to_integer(shift_val));
        Dout <= tmp_d;
```

```
      end if;
    end if;
  end process pr;

end my_shifter;
```

A.8 filter_template.vhd

```
-- -*-VHDL-*-
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
--use ieee.std_logic_arith.ALL;

entity filter is
    port ( X : in signed  (24-1  downto 0);
           Y : out signed (24-1  downto 0);
           reset: in STD_LOGIC;
           clk: in STD_LOGIC;
           overflow: out std_logic;
           chirpedge: in std_logic;
           preshiftIn: in unsigned (3 downto 0);
           preshiftOut: out unsigned (3 downto 0));
end filter;

architecture structure of filter is
  constant regbreite: integer := 24;

  component iir_filter0
    port ( Xpre : in signed  (regbreite-1  downto 0);
           Y : out signed (regbreite-1  downto 0);
           reset: in STD_LOGIC;
           clk: in STD_LOGIC;
           overflow_fpm: out std_logic);
  end component;
  component iir_filter1
    port ( Xpre : in signed  (regbreite-1  downto 0);
           Y : out signed (regbreite-1  downto 0);
           reset: in STD_LOGIC;
           clk: in STD_LOGIC;
           overflow_fpm: out std_logic);
  end component;

  signal Xpre, Y0, Y1, Y0_reg, Y1_reg: signed (regbreite-1 downto 0);
```

```
  signal ovf0,ovf1 : std_logic;

  signal preshift : unsigned(3 downto 0);

  signal chirpedge_old : std_logic;   -- um toggeln der chirpedge" zu erkennen
  signal ovfAnz : unsigned(13 downto 0);  -- zähler wie oft ein Overlow
                                -- während eines Schirps aufgetreten ist
  signal hugeAnz : unsigned(7 downto 0);  --Anzahl grosser Signale während
                                -- eines schirps

  signal overflow_fpm_local:   std_logic;     -- lokale kopie des output Signals

  constant max14Bit: unsigned(13 downto 0) := "11111111111111"; -- "11111111111111";
  -- !!! 14 Bit reichen um 3,3% der Samples zu zählen
  constant max8Bit: unsigned(7 downto 0) := "00001111";
  -- !!! 8 Bit reichen um 0,025% der Samples zu zählen

begin

  xpre <= shift_right(x, to_integer(preshiftIn));
  preshiftOut <= preshiftIn;

  bla0: iir_filter0 port map (xpre, y0, reset, clk, ovf0);
  bla1: iir_filter1 port map (Y0_reg, y1, reset, clk, ovf1);

  Y <= Y1_reg;
  overflow_fpm_local <= ovf0 or ovf1;
  overflow <= overflow_fpm_local;

  Y0_pro: process (clk, reset, Y0)
  begin
    if reset = '0' then
      Y0_reg <= (others => '0');
    elsif rising_edge(clk) then
      Y0_reg  <= y0;
    end if;
  end process y0_pro;

  Y1_pro: process (clk, reset, Y1)
  begin
    if reset = '0' then
      Y1_reg <= (others => '0');
    elsif rising_edge(clk) then
      Y1_reg  <= y1;
    end if;
  end process y1_pro;
```

```
ovfCount: process (clk, reset)
begin
  if reset = '0' then
    ovfAnz <= (others => '0');
  elsif rising_edge(clk) then
    if '1'=overflow_fpm_local then
      if ovfAnz /= max14Bit then
        ovfAnz<=ovfAnz+1;
      end if;
    end if;
    if chirpedge_old/=chirpedge then
      ovfAnz <= (others => '0');
    end if;
  end if;
end process ovfCount;

hugeCount: process (clk, reset)
begin
  if reset = '0' then
    hugeAnz <= (others => '0');
  elsif rising_edge(clk) then
    if (x(x'left) /= x(x'left-1)) then    -- wenn die obersten bits ausgelastet
      if hugeAnz /= max8Bit then
        hugeAnz<=hugeAnz+1;
      end if;
    end if;
    if chirpedge_old/=chirpedge then
      hugeAnz <= (others => '0');
    end if;
  end if;
end  process hugeCount;

AGC:process (clk, reset)
begin
  if reset = '0' then
    preshift <= "0101";      --(others => '0');
  elsif rising_edge(clk) then
    if chirpedge_old/=chirpedge then     -- auf signalwecjhesel reagieren
      chirpedge_old<=chirpedge;          --    - "  -
      if (ovfAnz = max8Bit) then
        if (preshift(3 downto 0) /= 15) then
          preshift <= preshift + 1;   --runtershiften
        end if;
      elsif (hugeAnz /= max14Bit) then
        if (preshift(3 downto 0)/= 0) then
          preshift <= preshift - 1;   --hochshiften
        end if;
```

```
        end if;
      end if;
    end if;
  end process AGC;

end structure;
```

A.9 stufe0.dat

```
#daten fuer stufe
# order ist die ordnung des filters
# ordergt2 = (order==2)?'--':''; ordereq2 = (order>=2)?'--':''
#
# bei der erstellung der koeffizienten muss darauf geachtet werden,
# dass komplementdarstellung verwendet wird. Die herkoemmlichen ausgaben
# von bc sind also nicht zu verwenden... (fuer neg. zahlen, ne)
#
**order**:2
**ordergt2**:--
**ordereq2**:
**real**:--
**sat**:
**nummer**:0
**regbreite**:24
**koeffbreite**:12
**beta0**: 1
**alphakoeffizienten**: "010000101010", "100000000000", "010000101010"
**betakoeffizienten**: "010000000000", "100001001001", "001110111001"
**preshiftr**:0
**nopreshiftr**:--
```

A.10 stufe1.dat

```
#daten fuer stufe
# order ist die ordnung des filters
# ordergt2 = (order==2)?'--':''; ordereq2 = (order>=2)?'--':''
**order**:2
**ordergt2**:--
**ordereq2**:
**real**:--
**sat**:
**nummer**:1
**regbreite**:24
**koeffbreite**:12
```

```
**beta0**:1
**alphakoeffizienten**: "010000000111", "100000000000", "010000000111"
**betakoeffizienten**: "010000000000", "100000011101", "001111100100"
**preshiftr**:4
```

A.11 bpig - ein Präprozessor

```
#!/usr/bin/perl

die "bpig bpig-datei template\n" unless $#ARGV == 1;

open IN, "cat $ARGV[0]|col|";           # bpig-script einlesen
@script = <IN>;
close IN;
while ($script[0]=~/^###/) { shift @script; }   # kommentare rauswerfen

foreach (@script)
       {
         if (/^\*\*([a-zA-Z0-9]+)\*\*:(.*)$/)
           {
             $marke=$1;
             $HASH{$marke} = $2;
           }
         else
           {
             $HASH{$marke} .= $_;
           }
       }

undef $/;                               # input record separator killen
open IN, "cat $ARGV[1]|col|";           # template lesen
$template = <IN>;
close IN;

foreach (keys %HASH)                    # alle marken ersetzen
       {
         $template=~s/\*\*$_\*\*/$HASH{$_}/g;
       }

print $template;
```

A.12 Makefile

```
VHDLOPT = -nc
VHDLAN = vhdlan $(VHDLOPT)
PREPRO = cpp -E -P
```

```
TEMPLATE_FILE = $(patsubst templates/%,%,$(wildcard templates/*.vhd))
SRC_FILE = shifter.vhd clk-div.vhd mul.vhd stufe0.vhd stufe1.vhd nios_wrap.vhd
CHIRP_FILE = chirp-gen1.vhd chirp-gen2.vhd

$(TEMPLATE_FILE):%.vhd: templates/%.vhd
-$(PREPRO) $< > $@

stufe0.vhd: template/stufe0.dat template/stufe_template.bpigvhd
bpig template/stufe0.dat template/stufe_template.pbigvhd >stufe0.vhd

stufe1.vhd: template/stufe1.dat template/stufe_template.bpigvhd
bpig template/stufe1.dat template/stufe_template.pbigvhd >stufe1.vhd


all: $(TEMPLATE_FILE) $(SRC_FILE) templates/$(TEMPLATE_FILE)
$(VHDLAN) $(TEMPLATE_FILE) $(CHIRP_FILE) $(SRC_FILE)

.PHONY: gen
gen: $(TEMPLATE_FILE)
```

Anhang B

Fotos, Schalt- und Bestückunspläne

Abbildung B.1: Ultraschallsensor

Abbildung B.2: Analogverstärker, AD- und DA-Wandler

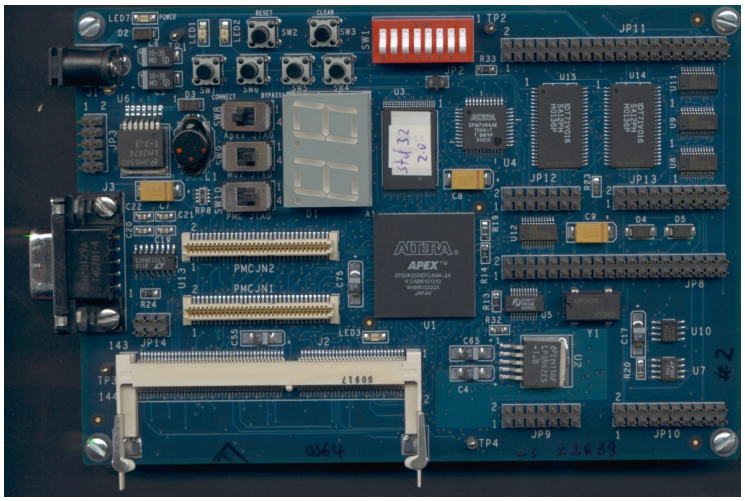


Abbildung B.3: Excalibur-Board

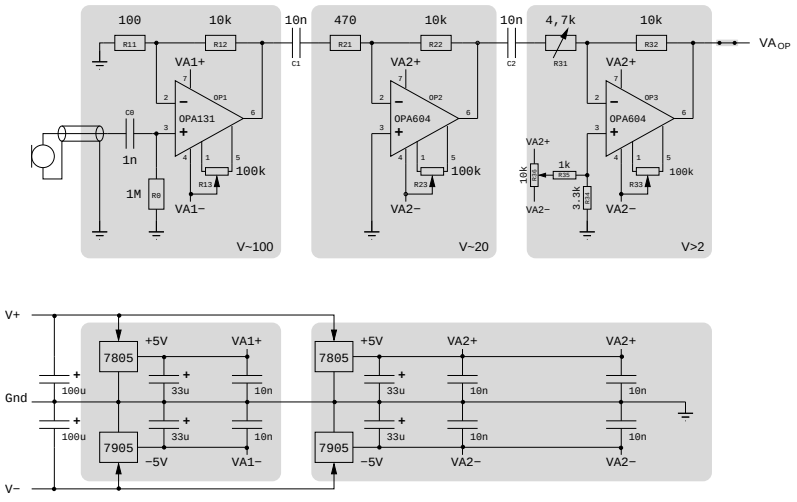


Abbildung B.4: Schaltplan des Analogverstärkers

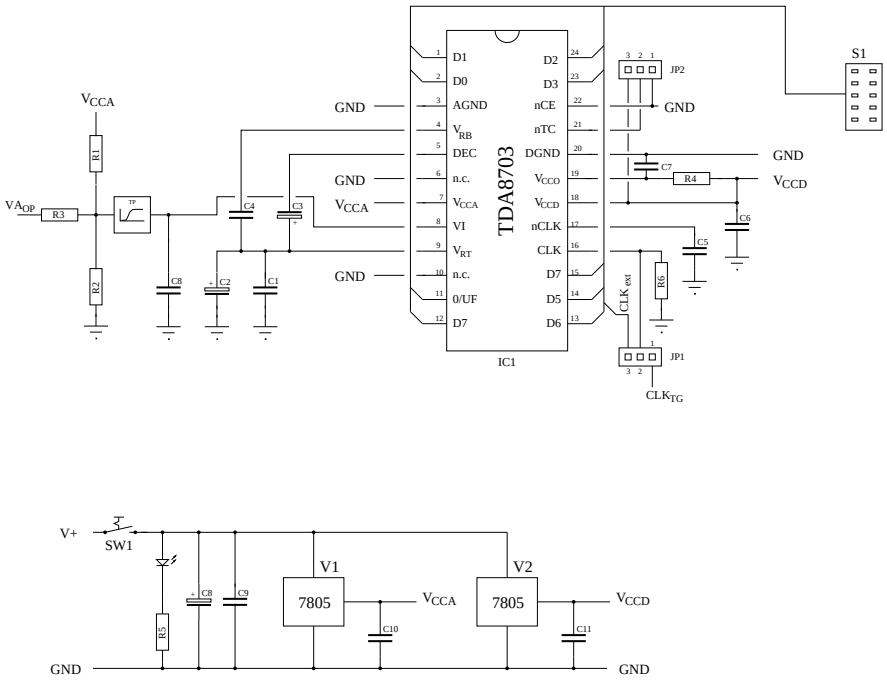


Abbildung B.5: Schaltplan des Analog-Digital-Wandlers

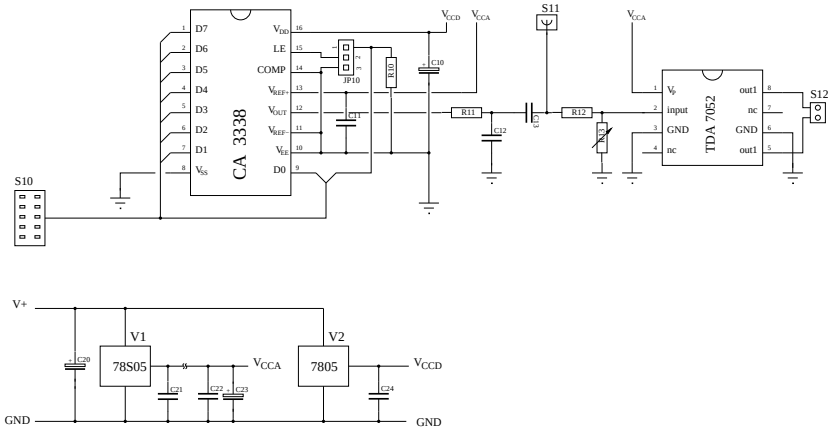


Abbildung B.6: Schaltplan des Digital-Analog-Wandlers

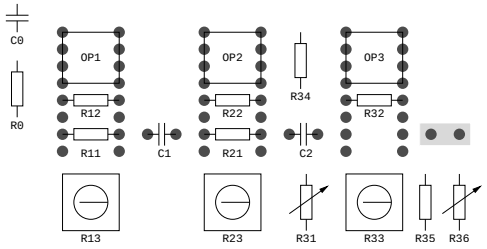
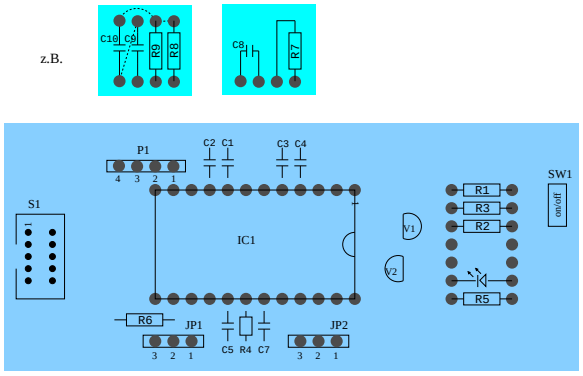


Abbildung B.7: Bestückungsplan des Analogverstärkers



S1:

Pin 1	D0
Pin 2	D1
Pin 3	D2
Pin 4	D3
Pin 5	D4
Pin 6	D5
Pin 7	D6
Pin 8	D7
Pin 9	O/UF
Pin 10	CLK_{ext}

JP1

2−1	CLK_{TG}
2−3	CLK_{ext}

JP2

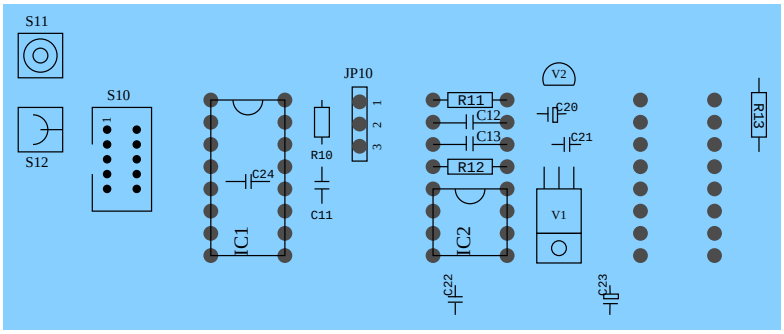
2−1	nTC = GND
2−3	nTC = V_{CCD}

P1

1	VA_{LS}
2	$IC1_{VI}$
3	$IC1_{VI}$
4	GND

				C1	22n
IC1	TDA8703	R1	1.5k	C2	4.7µ
		R2	10k	C3	4.7µ
		R3	2.2K	C4	47p
		R4	22Ω	C5	100p
		R5	3.3K	C6	100p
		R6	10κ	C7	22n
		R7	1κ	C8	2.2n
V1	7805	R8	820Ω	C9	2.2n
V2	7805	R8	820Ω	C10	2.2n

Abbildung B.8: Bestückungsplan des Analog-Digital-Wandlers



S10:

Pin 1	D0
Pin 2	D1
Pin 3	D2
Pin 4	D3
Pin 5	D4
Pin 6	D5
Pin 7	D6
Pin 8	D7
Pin 9	JP10/1
Pin 10	GND

IC1	CA 3338E	R10	1K	C11	100n
IC2	TDA 7052	R11	1K	C12	47n
		R12	10K	C13	47n
		R13	2,2k		
V1	78S05			C20	47µ
V2	7805			C21	100n
				C22	100n
JP10				C23	100µ
2−1	CLK_{ext} (S10/9)			C24	100n
2−3	non−clocked operation				

Abbildung B.9: Bestückungsplan des Digital-Analog-Wandlers

Literaturverzeichnis

[TieSch] U. Tietze/Ch. Schenk: *Halbleiter-Schaltungstechnik*, Springer Verlag, *ISBN: 3-540-56184-6*

[Ash] Peter J. Ashenden: *The designer's Guide To VHDL*, Second Edition, Morgan Kaufmann Publishers, *ISBN: 1-55860-674-2*

[ReiSch] J. Reichardt/B. Schwarz: *VHDL-Synthese*, Oldenburg Wissenschaftsverlag, *ISBN: 3-486-25128-7*