Implementierung und Analyse einer digitalen Signalsynthese auf einer rekonfigurierbaren Recheneinheit

Implementation and Analysis of a Digital Signal Synthesis on a Reconfigurable Computing Unit

Bachelor-Thesis von Konrad Stahlschmidt aus Limburg a.d. Lahn

Oktober 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich 20 (Informatik)
FG Eingebettete Systeme und ihre Anwendungen

Implementierung und Analyse einer digitalen Signalsynthese auf einer rekonfigurierbaren Recheneinheit
Implementation and Analysis of a Digital Signal Synthesis on a Reconfigurable Computing Unit

Vorgelegte Bachelor-Thesis von Konrad Stahlschmidt aus Limburg a.d. Lahn

1. Gutachten: Prof. Dr. Andreas Koch
2. Gutachten: Dipl.-Inform. Andreas Engel

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 24. Oktober 2011

(K. Stahlschmidt)

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Rahmen dieser Bachelorarbeit wurde eine digitale Signalsynthese auf einer rekonfigurierbaren Recheneinheit implementiert und analyisiert. Bei der Signalsynthese handelt es sich um die *Direct Digital Synthesis* (DDS). Die Signalsynthese ist ein wichtiges Thema in der Elektrotechnik und Informationstechnik. Durch die zunehmende Verwendung digitaler Komponenten gewinnt die digitale Signalsynthese an Bedeutung.

Auf dem Elektronikmarkt sind bereits fertige DDS Implementierungen auf einem Chip zu kaufen. [Anal99] Dennoch lohnt es sich, eine DDS auf einer rekonfigurierbaren Recheneinheit zu implementieren, weil sich damit Hardware-Kosten sparen lassen und eigene Anforderungen besser anpassen lassen. Ein bereits bestehendes System kann um eine DDS erweitert werden, sodass die Signalerzeugung parallel zur eigentlichen Anwendung läuft.

Diese DDS Implementierung soll verwendet werden, um Piezo-Elemente anzutreiben. Durch den piezoelektrischen Effekt wird eine Verformung von Materialien ermöglicht, wenn eine elektrische Spannung angelegt wird. [Aure01] Mithilfe der Piezo-Elemente sollen Motoren angetrieben werden.

1.2 Anforderungen

Die DDS soll auf einem Field Programmable Gate Array (FPGA) implementiert werden. Mithilfe einer DDS können beliebige periodische Signalformen dargestellt werden. Gefordert wird ein Sinussignal. Dabei sollen die Frequenz, Amplitude und Phasenverschiebung zur Laufzeit einstellbar sein. Eine Ausgangsspannung von -10 V bis $+10$ V wird gefordert. Der Frequenzbereich soll variierbar sein. Für das Schwingen der Piezo Elemente wird eine Frequenz zwischen 130 kHz und 150 kHz benötigt. Die Frequenzauflösung soll beliebig einstellbar sein. Die DDS soll eine beliebige Anzahl an Kanälen betreiben können. Für die Motoransteuerung werden drei Kanäle benötigt. Eine Klirrfaktor-Minimierung wird angestrebt.

2 Verfahren zur Sinuserzeugung

Es gibt verschiedene Verfahren zur Erzeugung eines Sinussignals. Im Folgenden werden grundlegende Prinzipien vorgestellt, um dieses elementare Signal zu generieren. Die unterschiedlichen Verfahren weisen allesamt sowohl Vor- als auch Nachteile auf und sollten je nach Anwendung eingesetzt werden.

2.1 LC-Schwingkreis

Ein LC-Schwingkreis stellt eine einfache Variante mit sehr geringem Aufwand dar, um ein Sinussignal zu erzeugen. In der Theorie werden lediglich zwei Bauteile gebraucht. Ein LC-Schwingkreis wird durch eine Schaltung realisiert, die aus einer Spule (L) und einem Kondensator (C) besteht. Die Bauteile können sowohl in Reihe, als auch parallel geschaltet werden. In Abbildung 2.1 ist die Schaltung eines LC-Parallelschwingkreises

Abbildung 2.1.: LC-Parallelschwingkreis

dargestellt. Die Schaltung stellt das theoretische Prinzip dar und beachtet dabei keine Verluste.

In dem Schwingkreis wird periodisch Energie zwischen dem magnetischen Feld der Spule und dem elektrischen Feld des Kondensators ausgetauscht, wodurch eine periodische Schwingung entsteht. Die Frequenz, mit der die Schaltung schwingt, ist die Resonanzfrequenz, die mithilfe der Thomsonschen Schwingungsgleichung

$$f_R = \frac{1}{2\pi\sqrt{LC}} \tag{2.1}$$

berechnet werden kann. [Ohmb11]

In der Praxis haben alle Bauteile Verluste. Kondensatoren und Spulen besitzen ohmsche Widerstände, die Verluste verursachen. Leitungen und Kontaktflächen sind die Ursachen für diese Widerstände. Bei Spulen wird der ohmsche Widerstand durch die vorhandenen Drahtwindungen stark beeinflusst. Außerdem werden Verluste durch abgestrahlte elektromagnetische Wellen erzeugt. Diese entstehen durch eine nicht vollständig isolierende Außenschicht an den Bauteilen. Die Verluste beeinflussen das Verhalten der Komponenten. Sie verursachen eine abweichende Ausgangsfrequenz.

Des Weiteren haben alle Bauteile Toleranzen, die vom Hersteller angegeben werden. Je nachdem, welche Fertigungsqualität verwendet wurde, können die Toleranzen zwischen 1 % und 10 % betragen. Die abweichenden Bauteilwerte beeinflussen ebenfalls die Resonanzfrequenz.

Neben den Bauteiltoleranzen und den Verlusten kann auch die Temperatur das Verhalten der Komponenten beeinflussen. Bei hohen oder niedrigen Temperaturen weisen die Bauteile in der Regel andere Werte auf, als bei Zimmertemperatur. Gerade die Eigenerwärmung der Bauteile kann während des laufenden Betriebs die Ausgangsfrequenz verändern.

Des Weiteren würde der Schwingkreis, wenn er einmal eingeschwungen ist, wegen der Dämpfung bzw. der ohmschen Widerstände, nur eine bestimmte Zeit schwingen und dabei abklingen. Aus diesem Grund werden aktive Verstärkerschaltungen verwendet, um dem Schwingkreis die benötigte Energie zu liefern, sodass die Verluste der ohmschen Widerstände kompensiert werden. [Brau04]

Für eine veränderbare Resonanzfrequenz ist ein variabler Kondensator und/oder eine variable Spule erforderlich. Für einen variablen Kondensator kann z.B. ein Drehkondensator verwendet werden. Bei einer Spule wird ein veränderlicher Wert erreicht, indem ein frei beweglicher Kern benutzt wird, der in die Spule hinein- und hinausgeschoben werden kann. Dadurch lässt sich die Kapazität und/oder Induktivität verändern. Die Frequenz wird innerhalb einer bestimmten Bandbreite einstellbar.

2.2 RC-Oszillator

Um ein Sinussignal zu erzeugen, kann ein Oszillator verwendet werden. Ein Oszillator kann mit einem Verstärker V (Verstärkung $\underline{V}$) und einer Rückkopplung k (Verstärkung $\underline{k}$) aufgebaut werden. Das System muss eine Selbstregelung beinhalten, mit der Verluste ausgeglichen werden. Abbildung 2.2 zeigt das Blockschaltbild eines Oszillators.

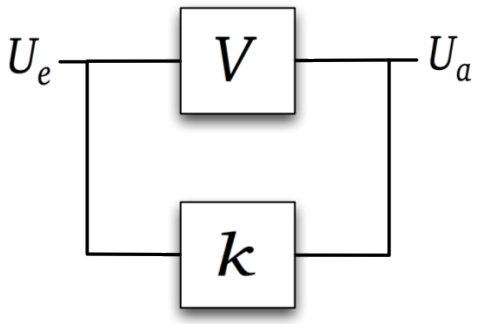


Abbildung 2.2.: Oszillator System

V und k sind beliebige Übertragungsfunktionen mit frequenzabhängiger Amplitudenverstärkung und Phasenverschiebung. Für ein schwingungsfähiges System muss die Schleifenverstärkung $\underline{k} \cdot \underline{V} \geq 1$ sein (Schwingbedingung). Wenn die Phasenverschiebung für einen Schleifenumlauf bei einer Frequenz $\phi_k + \phi_V = 0$ bzw. ein Vielfaches von 360° ist (Phasenbedingung), entsteht eine eigenständige Schwingung in genau dieser Frequenz (Oszillatorfrequenz).

Um die Oszillatorfrequenz zu bestimmen, werden frequenzabhängige Netzwerke in dem rückgekoppelten System eingesetzt. Diese Netzwerke erfüllen die Phasenbedingung bei genau einer Frequenz. Es können z.B. RC-Glieder, Schwingkreise oder Quarze eingesetzt werden. [Schm11b]

Abbildung 2.3 zeigt die Schaltung eines RC-Oszillators. (Vgl.: [Hein11], [Schm11b], [Lehw11])

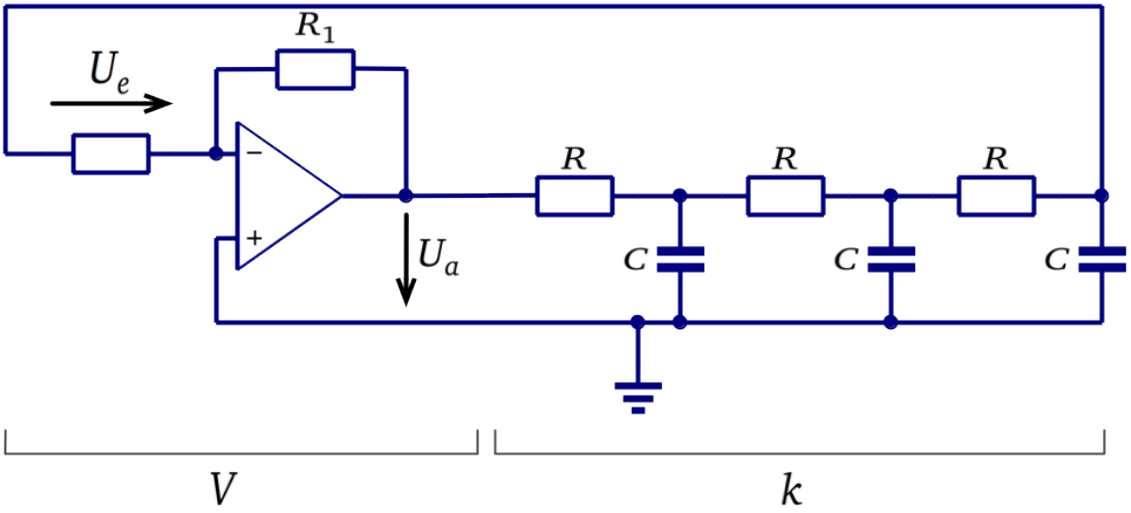


Abbildung 2.3.: RC-Oszillator Schaltung

Jedes RC-Glied erzeugt eine Phasenverschiebung zwischen 0° und 90°. Somit sind in dem RC-Phasenschiebernetzwerk mindestens drei gleich dimensionierte RC-Glieder notwendig, um die Phase um 180° zu drehen. Jedes RC-Glied dreht die Phase dann um 60°. [Miet11] Bei $\phi_k = -180°$ wird die Phasenbedingung erfüllt, weil der Operationsverstärker durch den invertierenden Anschluss die Phase um $\phi_V = 180°$ dreht. [Schm11b]

Die Oszillatorfrequenz ergibt sich nach [Lehw11] zu:

$$f_O = \frac{\sqrt{6}}{2\pi \cdot RC} \tag{2.2}$$

Der Verstärker muss die Abschwächung durch den Rückkopplungskreis ausgleichen. Der theoretische Verstärkungsfaktor muss nach [Miet11] bei $\underline{V} = 29$ liegen. Schaltungssimulationen haben gezeigt, dass der Faktor etwas über 29 liegen muss ($\underline{V} \geq 31$ bei $R_1 = 10\ \mathrm{M\Omega}$), damit eine ungedämpfte Schwingung erzeugt wird. Die Zeit, bis die Amplitude des Signals 95 % des Maximums erreicht hat (Einschwingzeit [Stef07]), beläuft sich in der Simulation auf ca. 200 ms.

RC-Oszillatoren benötigen keine Spulen, weshalb sie in tieferen Frequenzbereichen bis zu einigen MHz verwendet werden. Andere Oszillatoren benötigen bei diesen Frequenzen hohe Induktivitäten, welche viel Platz verbrauchen und teuer sind. [Meli07]

2.3 Rechteck als Grundfunktion

Ein Sinussignal lässt sich auch mit einem Rechtecksignal als Grundfunktion generieren. Das hat den Vorteil, dass die Frequenz leicht einstellbar ist, weil ein Rechtecksignal mit einfachen Mitteln und mit einer variierbaren Frequenz erzeugt werden kann. Das Rechtecksignal lässt sich z.B. mit einem astabilen Multivibrator erzeugen. Dabei wechselt die Ausgangsspannung der Schaltung periodisch zwischen den Zuständen *High* und *Low*. Neben Transistor-Entwürfen gibt es kommerzielle ICs (Integrated Circuits), die dieses Verhalten aufweisen. Bekannt ist vor allem der Timerbaustein *NE555*.

Bei dieser Methode zur Generierung eines Sinussignals wird ein Tiefpassfilter (TPF) benötigt. Dieser ist zwingend notwendig und entscheidet maßgebend die Qualität des Ausgangssignals. Es folgt ein kleiner Einblick in die Frequenztheorie, um zu erläutern, wieso ein TPF ein Rechtecksignal in ein Sinussignal wandeln kann.

Ein ideales, zeitsymmetrisches Rechtecksignal weist unendlich viele Frequenzanteile auf. Je höher die Oberwelle, desto kleiner ist die Amplitude dieser Welle. Die Wellen (Grundwelle, Oberwellen) in einem Frequenzspektrum sind einzelne Sinussignale, die in einem Rechteck überlagert vorkommen. Wir betrachten die Fouriertransformierte eines Rechtecksignals in Gleichung 2.3. [KrSk08]

$$F(t) = \frac{4A}{\pi} \cdot \sum_{k=0}^{\infty} \frac{\sin((2k+1)\omega t)}{2k+1} \tag{2.3}$$

Die Variable A bestimmt die Amplitude des Rechtecksignals, k den Grad der Oberwelle. Die Gleichung verdeutlicht, dass ein ideales Rechtecksignal durch unendlich viele Sinussignale dargestellt werden kann. Ein steigendes k verursacht einen kleineren Funktionswert, was die kleinere Amplitude bei höheren Oberwellen erklärt.

Für die Erzeugung eines Sinussignals ist nur die Grundwelle von Interesse und man versucht, die Oberwellen zu unterdrücken. Dazu wird ein Tiefpassfilter verwendet. Bei einem Tiefpassfilter wird die sogenannte Grenzfrequenz f_g angegeben. Als Grenzfrequenz wird die Frequenz bezeichnet, bei der die Amplitude des Signals um das $\frac{1}{\sqrt{2}}$-fache gedämpft wird. Dies entspricht etwa -3 dB. Der Phasenwinkel beträgt bei der Grenzfrequenz 45 Grad. [Herb02] Frequenzen unterhalb der Grenzfrequenz gelten als durchgelassen. Frequenzen oberhalb von f_g gelten als gesperrt.

Abbildung 2.4 zeigt das Frequenzspektrum eines Rechtecksignals mit einer Grundfrequenz f_0.

Die Grundwelle liegt bei $1 \cdot f_0$. Die erste Oberwelle befindet sich bei einem idealen Rechtecksignal bei $3 \cdot f_0$. Die weiteren Oberwellen liegen bei $5 \cdot f_0$, $7 \cdot f_0$, usw. Die rote Linie in der Abbildung stellt exemplarisch die Dämpfung eines Tiefpassfilters dar und beträgt 80 dB pro Dekade. Die Grundwelle wird nicht bzw. nur sehr wenig gedämpft. Die erste Oberwelle dagegen wird bereits um 36 dB gedämpft und die Amplitude der zweiten Oberwelle wird mit 54 dB stark gedämpft. Alle Amplituden der höheren Oberwellen kommen sehr stark gedämpft im Ausgangssignal vor. Da die Grundwelle des Rechtecksignals bereits ein ideales Sinussignal darstellt, wächst die Signalqualität mit steigender Flankensteilheit des Tiefpassfilters. [Weiß05] [Herb02]

Die Theorie von Tiefpassfiltern und wie die Flankensteilheit verbessert werden kann, wird in Kapitel 3.5 erläutert.

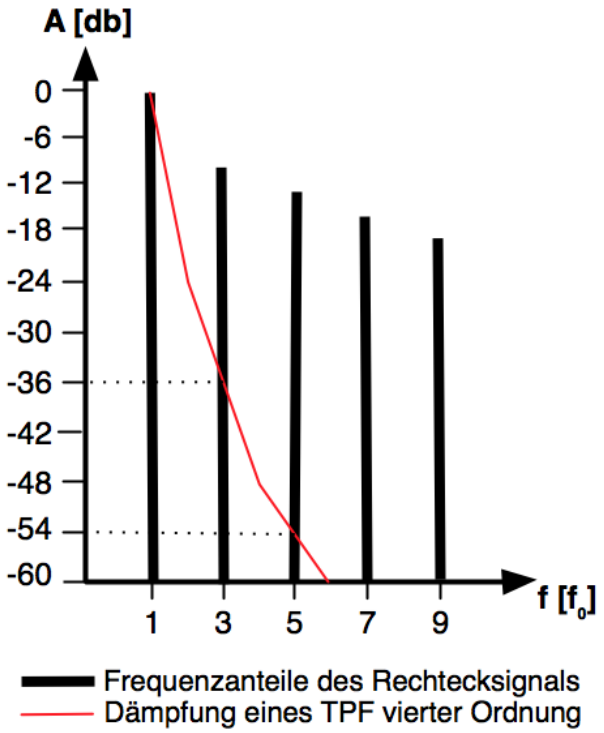


Abbildung 2.4.: Frequenzspektrum eines Rechtecksignals

Wenn ein Sinussignal mit einer variablen Frequenz erzeugt werden soll, ist dies mithilfe *eines* Tiefpassfilters nur bedingt möglich. Zwischen der Grundwelle und der ersten Oberwelle liegt eine Bandbreite von Δf (bei einem Rechtecksignal $\Delta f = 2 \cdot f_0$). In dieser Bandbreite muss die Grenzfrequenz des TPF liegen. Wenn eine größere Bandbreite benötigt wird, wäre entweder eine variable Grenzfrequenz oder mehrere Tiefpassfilter nötig. Bei dem Einsatz von mehreren TPFs müsste je nach eingestellter Frequenz der zugehörige TPF verwendet werden.

2.4 Direct Digital Synthesis

Die Direct Digital Synthesis ist eine aktuelle Technik zur Erzeugung beliebiger periodischer Signale. Im Unterschied zu den bisher vorgestellten Signalgeneratoren wird das Signal bei der DDS im Wesentlichen durch eine digitale Schaltung generiert.

2.4.1 Prinzip

Die Direct Digital Synthesis arbeitet mit einem Akkumulator, einem Amplituden-Generator, einem Digital-Analog-Konverter (engl.: Digital-to-Analog-Converter, DAC) und einem (optionalen) Tiefpassfilter.

Abbildung 2.5 zeigt den schematischen Aufbau. Nach jedem System-Block sind die zugehörigen Ausgangssignale bei der Erzeugung eines Sinussignals skizziert. Durch den im Phasen-Akkumulator gespeicherten Wert wird die Amplitude ausfindig gemacht, welche anschließend an die gewünschte Amplitudenskalierung angepasst und

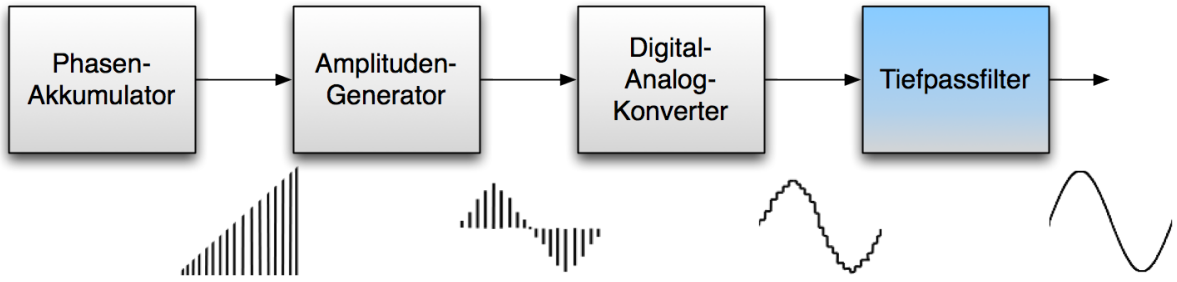


Abbildung 2.5.: Blockschaltbild der Direct Digital Synthesis

an den Digital-Analog-Konverter weitergegeben wird. Der DAC erzeugt aus dem anliegenden digitalen Wert den dazugehörigen Analogen. Die Werte der Amplituden liegen im Digitalen nur quantisiert vor. Nach einer Umwandlung in ein analoges Signal sind diese Quantisierungsfehler durch die entstandenen Stufen sichtbar. Zur Glättung des Signals kann ein Tiefpassfilter nachgeschaltet werden. Am Ausgang des TPF befindet sich ein Sinussignal mit abgeschwächten Stufen. [Stan11] Das Thema Tiefpassfilter wird in Kapitel 3.5 näher erläutert.

Die Schrittweite des Phasen-Akkumulators entscheidet über die Frequenz des Ausgangssignals. In Abbildung 2.6 ist die Phase eines periodischen Signals in Form eines Kreises dargestellt (Vgl. [Anal99]). Jeder der *N* Punkte entspricht einem Phasenwert. Wenn der Kreis einmal umlaufen wird, ist eine Periode vorüber.

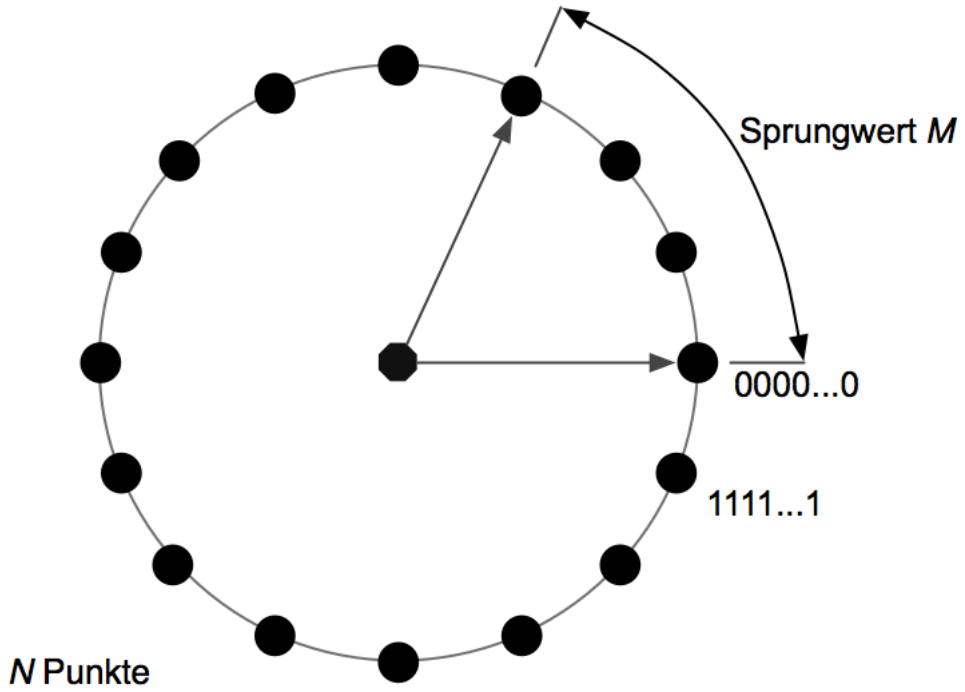


Abbildung 2.6.: Periode anhand des Phasen-Kreises

Der Phasen-Akkumulator addiert in jedem Takt einen Sprungwert *M*, sodass der Kreis umlaufen und die Periode abgearbeitet wird. Bei einem größeren Sprungwert ist die Periode eines Signals schneller beendet. Der Wert *M* bestimmt damit direkt die

Frequenz des Ausgangssignals. Durch die Anzahl der Phasenwerte wird die mögliche Genauigkeit der Ausgangsfrequenz beeinflusst. Wenn mehr Phasenwerte vorhanden sind, kann eine exaktere Frequenz eingestellt werden.

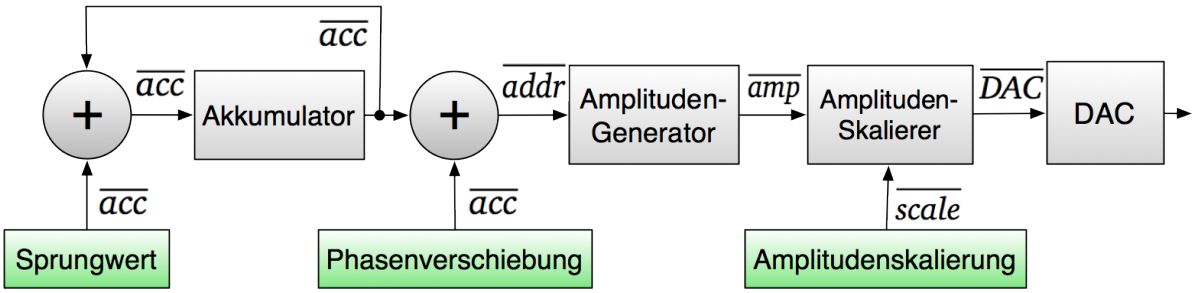
Die Frequenz des Ausgangssignals f_{out} lässt sich mithilfe der Akkumulator-Frequenz f_{acc}, der Anzahl der Stützstellen N (die Anzahl der Punkte bzw. Phasenwerte) und des Sprungwertes M berechnen. f_{acc} bezeichnet die Frequenz, mit welcher der Akkumulator aktualisiert wird. Die Anzahl der Phasenwerte N ist beliebig. Jedoch bietet es sich auf digitalen Systemen an, eine Zweierpotenz zu wählen. Nur dann werden die verbrauchten Ressourcen vollständig genutzt und es wird keine extra Logik zur Überlauferkennung benötigt. Es ergibt sich $N = 2^{\overline{N}}$, wobei $\overline{N}$ die Bitbreite des Akkumulators darstellt. Gleichung 2.4 beschreibt die Berechnung der Ausgangsfrequenz. [Anal99]

$$f_{out} = \frac{f_{acc} \cdot M}{N} = \frac{f_{acc} \cdot M}{2^{\overline{N}}} \tag{2.4}$$

Wenn eine Ausgangsfrequenz gegeben ist, wird der Sprungwert M benötigt. Dieser wird durch Umstellen von Gleichung 2.4 berechnet:

$$M = \frac{f_{out} \cdot 2^{\overline{N}}}{f_{acc}} \tag{2.5}$$

Zu jedem der N Phasenwerte muss der Amplituden-Generator den entsprechenden Funktionswert des Ausgangssignals bereit stellen. Dies lässt sich im einfachsten Fall mit einer Look-Up Table (LUT) ermöglichen. In der LUT müssen die Funktionswerte fortlaufend gespeichert sein. Die Quantisierungsfehler können durch breitere und tiefere LUTs reduziert werden.



$\overline{acc}$ = Bitbreite des Akkumulators
$\overline{addr}$ = Bitbreite der Adresse
$\overline{amp}$ = Bitbreite der Amplitude
$\overline{scale}$ = Bitbreite der Amplitudenskalierung
$\overline{DAC}$ = Bitbreite des DACs

Abbildung 2.7.: Datenpfad der Direct Digital Synthesis

Abbildung 2.7 zeigt den konkreten Datenpfad einer DDS Implementierung mit konfigurierbaren Bitbreiten.

Die Eingangsvariablen *Sprungwert*, *Phasenverschiebung* und *Amplitudenskalierung* sind grün hinterlegt. Der *Akkumulator* wird durch den rückgekoppelten aktuellen Wert und dem *Sprungwert* gebildet. Der zweite Addierer addiert die *Phasenverschiebung* auf den im *Akkumulator* gespeicherten Wert. Die höchst signifikanten $\overline{addr}$ Bits werden als Adresse an den *Amplituden-Generator* gelegt, der den dazugehörigen Funktionswert am Ausgang bereitstellt. Der *Amplituden-Generator* kann z.B. durch eine LUT realisiert werden. Weitere Möglichkeiten zur Amplitudengenerierung werden in Kapitel 3.3 erläutert. Die restlichen Bits des Akkumulators haben keinen direkten Einfluss auf das Ausgangssignal. Sie werden dennoch benötigt, um die Genauigkeit der Ausgangsfrequenz zu gewährleisten. Der *Sprungwert* und die *Phasenverschiebung* haben eine festgelegte Bitbreite von $\overline{acc}$. So lassen sich der Sprungwert und die Phasenverschiebung mit einer maximalen Genauigkeit einstellen. Auf die Bestimmung der Bitbreiten wird in Kapitel 3.2 näher eingegangen. Die am Ausgang des *Amplituden-Generator*s anliegenden Daten werden entsprechend der gewünschten Amplitude skaliert. Die Angabe der *Amplitudenskalierung* erfolgt mit einer einstellbaren Bitbreite von $\overline{scale}$. Die angepassten Werte werden an den DAC weitergegeben. Am Ausgang liegt das analoge Signal vor.

2.4.2 Bewertung

Eine DDS erlaubt es, die Genauigkeit des zu erzeugenden Signals sehr flexibel einzustellen. Die Genauigkeit der analogen Signalsyntheseverfahren ist hingegen durch die Bauteiltoleranzen der verwendeten Komponenten sowie durch deren temperatur- und alterungsabhänigen Eigenschaften begrenzt. Durch das Einstellen der Frequenz, Phasenverschiebung und Amplitudenskalierung ist die DDS in vielen Bereichen einsetzbar. Komplexe Signalgeneratoren sind bei analoge Verfahren nur mit hohem Schaltungsaufwand möglich.

Ausserdem lassen sich beliebige Signale mit dem selben Schaltungsaufbau realisieren. Hierbei muss lediglich der Amplituden-Generator angepasst werden. [Brau04] [Anal99]

Durch die digitale Kontrolle des Ausgangs lässt sich die DDS sehr gut in andere digitale Systeme einbinden und von diesen nutzen.

Auf einer MCU (Micro-Controller-Unit) wird für die Implementierung einer DDS regelmäßig Rechenzeit benötigt. Dafür wäre die Verwendung von Interrupts nötig, wenn noch etwas anderes auf dieser Recheneinheit berechnet werden müsste. Durch diese andauernden Kontextwechsel ginge viel Zeit verloren. Deshalb ist die Verwendung einer Hardware-Implementierung besser geeignet. Die Genauigkeit bzw. Qualität des Ausgangssignals hängt stark von der Größe der verwendeten Register ab und beeinflusst somit direkt die benötigten Hardware-Ressourcen. Gerade bei beschränkten Hardware-Komponenten muss eine hardwareschonende Implementierung vorhanden sein.

3 DDS Implementierung

Die Verilog-Implementierung der DDS, der Ansteuerung eines geeigneten DAC sowie der verwendeten Testumgebung liegt dieser Arbeit bei.

3.1 Modulhierarchie

Die Direct Digital Synthesis wurde so aufgebaut, dass Änderungen oder Erweiterungen leicht implementierbar sind. Es gibt ein Hauptmodul *DDS*, das alle nötigen Untermodule instanziiert. Dieses muss mit einem DAC-Controller verbunden werden, der über ausreichend Ausgabekanäle verfügt. Die Modulhierarchie kann in Abbildung 3.1 eingesehen werden.

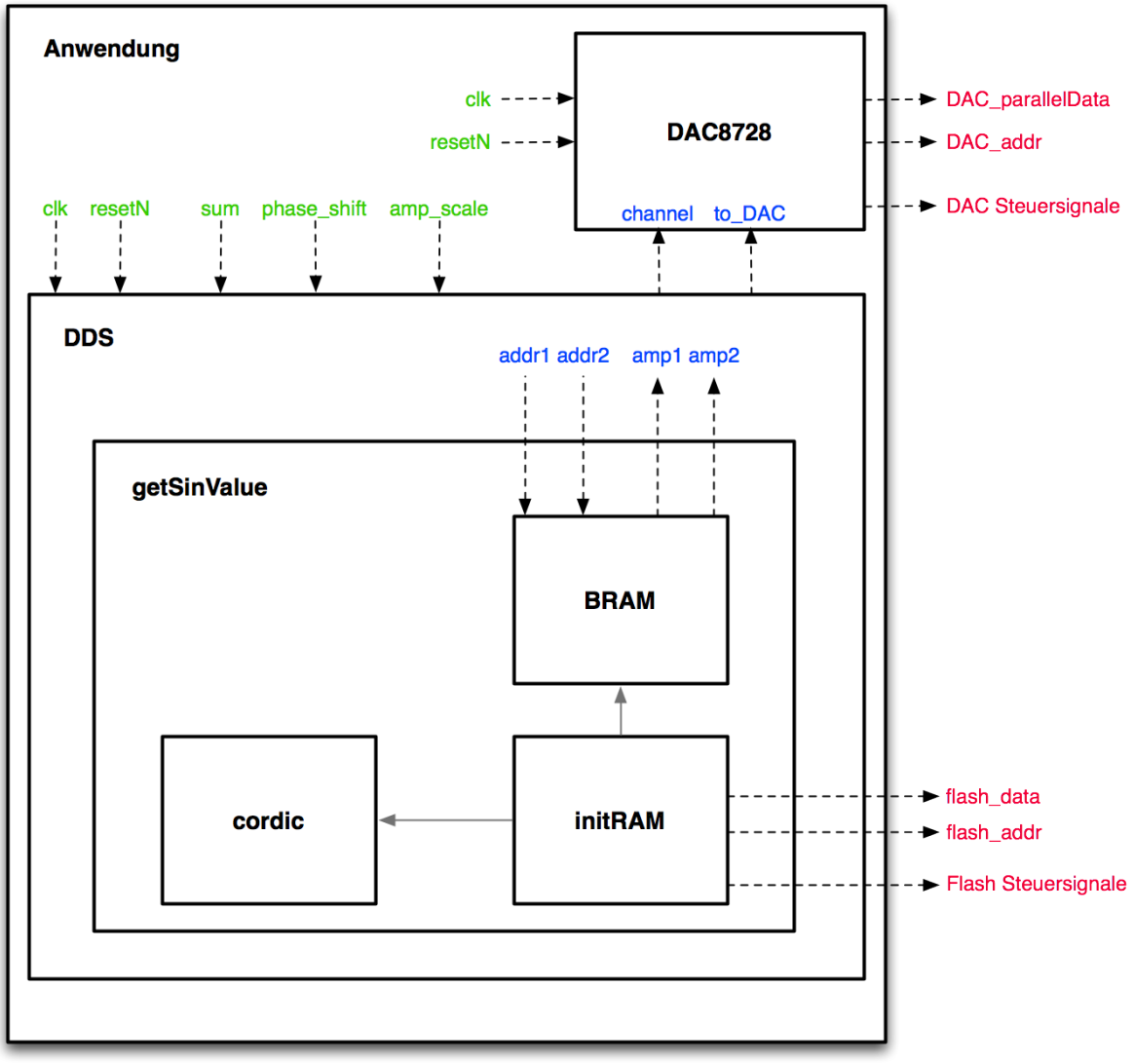


Abbildung 3.1.: Modulhierarchie

Die Modulhierarchie beinhaltet die wichtigsten Signalbezeichnungen. Alle Signale in grüner Schriftfarbe sind anwendungsspezifisch (Eingangssignale). Die blauen Bezeichner repräsentieren Werte, die von der *DDS* erzeugt bzw. verwendet werden. Die rot dargestellten Signale sind Ausgabewerte, welche mit der Hardware verbunden werden sollten. Die *DDS* selbst ist so aufgebaut, dass sie die Phasenwerte verwaltet und die Adressen an ein Modul *getSinValue* weitergibt. Hier wird der entsprechende Amplitudenwert ermittelt und der *DDS* mitgeteilt. Durch das Separieren des Amplituden-Generators ist es für die *DDS* nicht wichtig, wie die Amplitude ermittelt wurde. Lediglich die Dauer zum Abrufen eines Wertes (Anzahl Taktyzklen) muss bekannt sein. In der abgebildete Modulhierarchie wird ein Block-RAM (BRAM) als LUT genutzt. Kapitel 3.3 erläutert die Möglichkeiten zum Abrufen der Amplituden.

Eine weitere Flexibilität wird durch ein Modul *initRAM* erreicht. Dort wird der BRAM während der Initialisierung mit Amplitudenwerten gefüllt. Das Kapseln dieses Moduls ermöglicht eine leichte Erweiterung von unterschiedlichen Initialisierungsmethoden.

Die angegeben Signale und Module werden im Laufe der Arbeit diskutiert.

3.2 Erzeugung des Phasensignals

Der Akkumulator wird im *DDS*-Modul aktualisiert. Das Phasensignal wird durch die höchst signifikanten $\overline{addr}$ Bits des Akkumulators dargestellt. Das Akkumulator-Register benötigt eine bestimmte Größe, um die gewünschte Frequenzauflösung zu erreichen.

In dieser Implementierung ist die Anzahl der Ausgabekanäle frei wählbar. Jeder Kanal benötigt seinen eigenen Akkumulator. Bei mehreren aktiven Kanälen werden die Akkumulatoren mit denselben Ressourcen aktualisiert. Dies geschieht gezeitmultiplext alle $STEPS\_DAC \cdot CHANNEL\_AMOUNT$ Taktzyklen. Um weitere Ressourcen zu sparen, werden die notwendigen Registerbreiten der Akkumulatoren und der Sprungwerte berechnet. So lassen sich nur die gewünschten Anforderungen erfüllen und es werden keine Bits bzw. Ressourcen verschwendet.

Bei der Initialisierung wird mit zwei Frequenzen (*FREQ_OUT_LOW*, *FREQ_OUT_HIGH*) das Frequenzband spezifiziert, innerhalb dessen alle Ausgabekanäle variiert werden sollen. Deshalb werden auf den Akkumulator der Sprungwert, welcher zu der unteren Frequenz *FREQ_OUT_LOW* gehört, und der Summand addiert, der individuell für jeden Kanal ist. Über diesen als Eingabeport variierbaren Summanden lässt sich die Ausgangsfrequenz innerhalb der Frequenzbandbreite einstellen.

Die Frequenzauflösung kann per Parameter *FREQ_OUT_RESOLUTION* angepasst werden. Diese Konfiguration beeinflusst die Breite der verwendeten Register und Addierer und wirkt sich damit direkt auf die benötigten Hardwareressourcen aus, wie im Folgenden gezeigt wird.

Der Akkumulator (N, Bitbreite des Akkumulators $\overline{N}$) soll so groß sein, dass zu jedem Takt von f_{acc} eine minimale Auflösung von Q [Hz] addiert werden kann. Diese Größe wird durch die Zuweisung aus Gleichung 3.1 beschrieben. [Xili05]

$$N := \frac{f_{acc}}{Q} \tag{3.1}$$

Die Akkumulatorgröße wird in der Implementierung durch Registerbreiten bestimmt. Deshalb sollte der Maximalwert des Akkumulators eine Zweierpotenz sein. Wenn der Akkumulator größer ist, als in Gleichung 3.1 definiert, wird die mögliche Auflösung kleiner. Dies ermöglicht ein genaueres Konfigurieren der Ausgangsfrequenz und erfüllt somit die mindest geforderte Auflösung Q. Die Bitbreite des Akkumulators lässt sich dann folgendermaßen berechnen:

$$N \geq \frac{f_{acc}}{Q} \tag{3.2}$$

$$\overline{N} \geq log_2(\frac{f_{acc}}{Q}) \tag{3.3}$$

$$\overline{N} = \lceil log_2(\frac{f_{acc}}{Q}) \rceil \tag{3.4}$$

Die Bitbreite für den übergebenen Summanden (*sum*, Breite des Summanden $\overline{sum}$) lässt sich mithilfe der bereits bestimmten Akkumulatorgröße berechnen. Dabei wird Gleichung 2.4 aufgegriffen. Der Sprungwert M wird durch den Sprungwert der unteren Grenze des Frequenzbandes (*base*) und dem individuellen Sprungwert des Kanals bestimmt. Er ergibt sich also zu $M := base + sum$.

$$f_{out} = f_{acc} \cdot \frac{base + sum}{2^{\overline{N}}} \tag{3.5}$$

Gleichung 3.5 wird umgestellt, um *sum* zu bestimmen:

$$sum = \frac{f_{out} \cdot 2^{\overline{N}}}{f_{acc}} - base \tag{3.6}$$

sum muss gerade so groß sein, dass $sum + base$ das gesamte Frequenzband ($\Delta f_{out} :=$ *FREQ_OUT_HIGH* $-$ *FREQ_OUT_LOW*) abdeckt. Es folgt Gleichung 3.7.

$$sum = \frac{\Delta f_{out} \cdot 2^{\overline{N}}}{f_{acc}} \tag{3.7}$$

Der Wert für *sum* aus Gleichung 3.7 ist die minimal benötigte Größe. Deshalb darf die Größe für die Bitbreite aufgerundet werden und es ergibt sich für den Summanden:

$$\overline{sum} \geq \log_2(\frac{\Delta f_{out} \cdot 2^{\overline{N}}}{f_{acc}}) \tag{3.8}$$

$$\overline{sum} = \lceil \log_2(\frac{\Delta f_{out} \cdot 2^{\overline{N}}}{f_{acc}}) \rceil \tag{3.9}$$

Mit den Gleichungen 3.4 und 3.9 werden aus der konfigurierten Akkumulatorfrequenz, der Frequenzauflösung und dem Zielfrequenzband die minimalen Registerbreiten für den Akkumulator und den zur Laufzeit variierbaren Summanden bestimmt. Die

einstellbare Phasenverschiebung hat ebenfalls eine Bitbreite von $\overline{N}$. Damit erreicht die Phasenverschiebung ihre maximal mögliche Einstellgenauigkeit.

Ein typisches Beispiel mit einem Frequenzband zwischen 130 kHz und 170 kHz bei einer Auflösung von 0,01 Hz und einer Akkumulator-Frequenz von 15 MHz würde folgende Bitbreiten liefern:

$$\overline{N} = \lceil log_2(\frac{15\text{ MHz}}{0,01\text{ Hz}}) \rceil \approx \lceil 30,4832 \rceil = 31$$

$$\overline{sum} = \lceil log_2(\frac{40\text{ kHz} \cdot 2^{\overline{31}}}{15\text{ MHz}}) \rceil \approx \lceil 22,4493 \rceil = 23$$

Die berechneten Bitbreiten sind aufgerundet. Daher kann eine Vollaussteuerung des Summanden eine Frequenz oberhalb des Frequenzbandes erzeugen. Außerdem entspricht eine Veränderung des Summanden um Eins nicht exakt einer Frequenzänderung von *FREQ_OUT_RESOLUTION*, sondern ggf. einer kleineren Frequenzänderung.

3.3 Umsetzung zwischen Phase und Amplitude

Diese DDS Implementierung unterstützt verschiedene Methoden, um aus der Phase die Amplitude zu erhalten. Die unterschiedlichen Methoden werden im Modul *getSinValue* umgesetzt.

Zum Einen lässt sich eine Look-Up Table verwenden. In dieser sind die Amplitudenwerte der geforderten Signalform gespeichert und werden zur Laufzeit ausgelesen. Weitere Details zu der LUT-Methode folgen in Kapitel 3.3.1. Neben der LUT-Methode kann die Amplitude auch zur Laufzeit mithilfe des CORDIC-Algorithmus berechnet werden, welcher in Kapitel 3.3.2 ausführlich behandelt wird.

Wenn die Amplitude zur aktuellen Phase vorliegt (*amp*), erfolgt die Anpassung an die eingestellte Amplitudenskalierung (*amp_scale*). Gleichung 3.10 beschreibt diese Skalierung.

$$to\_DAC = \frac{amp \cdot amp\_scale}{2^{WIDTH\_AMP\_SCALE}} \tag{3.10}$$

Zur Reduktion des Speicherbedarfs wird die Symmetrie des Sinussignals ausgenutzt und nur eine Viertelperiode in der LUT abgelegt. Durch Spiegelung dieser Informationen lässt sich das Signal vollständig rekonstruieren. Gleichung 3.11 gibt an, wieviele Stützstellen für eine Periode vorhanden sind.

$$allSamples = samplesInRAM \cdot 4 = 2^{DEPTH\_SAMPLES} \cdot 4 \tag{3.11}$$

Wie sich die unterschiedlichen Umsetzungen zur Generierung der Amplitude auf die Signalgüte und die verbrauchten Ressourcen auswirken, wird im Kapitel 4 behandelt.

3.3.1 LUT

Für eine LUT eignet sich z.B. ein ROM (Read Only Memory) oder BRAM, weil diese schnelle Zugriffs- und Antwortzeiten aufweisen. Ein Flash-Speicher sollte nicht als Look-Up Table verwendet werden. Der Stromverbrauch des verwendeten Flashs (ca. 50 mW [Numo07]) ist höher als bei dem FPGA internen BRAM (ca. 75 μW [Acte09]), was gerade im Dauerbetrieb einen Nachteil darstellt. Wenn die Taktfrequenz der DDS hoch genug ist, kann der Flash-Speicher die Amplituden nicht schnell genug ausgeben, sodass die DDS auf den Flash warten muss. Die Taktfrequenz müsste bei einem DAC, der zwei Taktzyklen zum Entgegennehmen eines Wertes benötigt, und einem Flash-Speicher, der 75 ns zum Ausgeben eines Wertes braucht, über $26,\overline{6}$ MHz liegen. Des Weiteren wird für eine später erläuterte Implementierungserweiterung (Kapitel 3.3.3) ein Dual-Port fähiger Speicher benötigt. Aus diesen Gründen wird ein BRAM als LUT genutzt.

Um eine Look-Up Table zu initialisieren, gibt es mehrere Möglichkeiten. Zwei davon werden hier vorgestellt.

Mithilfe eines externen Flash-Bausteins, in dem die Werte bereits vorhanden sind, können die Werte aus dem Flash-Speicher gelesen und in den BRAM geschrieben werden. Der Flash-Baustein speichert Daten dauerhaft auch ohne Stromversorgung und bedarf deshalb nur einer einmaligen Initialisierung, bei der die Werte z.B. auf dem PC berechnet werden können und dann auf den Flash-Speicher gespielt werden.

Außer einer Initialisierung mit einem Flash-Bausteins lässt sich der CORDIC-Algorithmus verwenden. Dabei werden die Amplituden allerdings nur einmal zur Initialisierung generiert und im RAM gespeichert.

3.3.2 CORDIC

Für die Hardware-Realisierung einer Sinus-Funktion ist insbesondere der CORDIC-Algorithmus (*COordinate Rotation DIgital Computer*) geeignet. Es lassen sich zur Laufzeit iterativ Funktionswerte von verschiedenen elementaren Funktionen (z.B. trigonometrische, exponentiale, ...) auf Basis von Vektorrotationen auf dem Einheitskreis berechnen. Dabei verwendet der CORDIC-Algorithmus eine geeignete Schrittweite zum Verschieben des Einheitsvektors, bis der gesuchte Winkel gefunden wird. [GaDS11] [Riss04] Die Funktionsweise wird im Folgenden beschrieben.

Abbildung 3.2 zeigt die Drehung eines Vektors (x_0, y_0) um den Winkel λ. Das Resultat ist der Vektor (x, y). Die Vektordrehung lässt sich durch Gleichung 3.12 und 3.13 beschreiben.

$$x = x_0 \cdot \cos(\lambda) - y_0 \cdot \sin(\lambda) \tag{3.12}$$

$$y = y_0 \cdot \cos(\lambda) + x_0 \cdot \sin(\lambda) \tag{3.13}$$

Um die Drehung um λ zu realisieren, wird bei dem CORDIC-Verfahren der Winkel λ durch Teilwinkel angenähert. Die Teilwinkel α_i sind fest vorgegeben. n bezeichnet

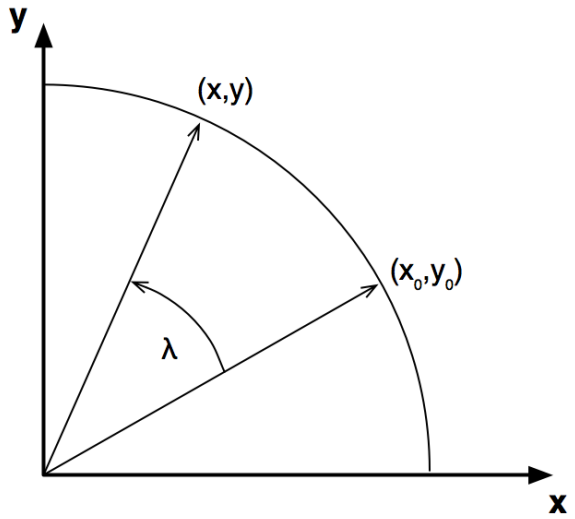


Abbildung 3.2.: Drehung eines Vektors um λ Grad

die Anzahl der Teildrehungen (Iterationen). Es ergibt sich der Zusammenhang, wie in Gleichung 3.14 dargestellt.

$$\lambda \approx \sum_{i=0}^{n-1} \sigma_i \alpha_i \qquad \text{mit } \sigma_i \in \{1, -1\} \tag{3.14}$$

σ_i gibt das Vorzeichen für die Teilwinkel an und sollte so gewählt werden, dass der Gesamtwinkel möglichst genau approximiert wird. Die Teilwinkel werden definiert zu:

$$\alpha_i := \arctan(2^{-i}) \tag{3.15}$$

Abbildung 3.3 zeigt die Verschiebung des Vektors (x_i, y_i) bei einem Teilschritt.

R_i ist der Betrag zu dem Vektor (x_i, y_i). Der Winkel des Vektors (x_i, y_i) wird durch ϕ_i repräsentiert. Der Winkel der Teildrehung beträgt α_i und besitzt das Vorzeichen σ_i.

Durch die neu eingeführten Variablen lässt sich die Vektordrehung in Polar-Koordinaten angeben, wie die Gleichungen 3.16 und 3.17 zeigen.

$$x_{i+1} = R_{i+1} \cdot \cos(\phi_i + \sigma_i \alpha_i) \tag{3.16}$$

$$y_{i+1} = R_{i+1} \cdot \sin(\phi_i + \sigma_i \alpha_i) \tag{3.17}$$

Es gelten außerdem die Beziehungen $x_i = R_i \cdot \cos(\phi_i)$ und $y_i = R_i \cdot \sin(\phi_i)$. Die Gleichungen 3.16 und 3.17 lassen sich mithilfe des Additionstheorems für trigonometrische Funktionen umformen:

$$x_{i+1} = R_{i+1} \cdot [\cos(\phi_i)\cos(\sigma_i \alpha_i) - \sin(\phi_i)\sin(\sigma_i \alpha_i)] \tag{3.18}$$

$$x_{i+1} = R_{i+1} \cdot [\frac{x_i}{R_i}\cos(\sigma_i \alpha_i) - \frac{y_i}{R_i}\sin(\sigma_i \alpha_i)] \tag{3.19}$$

$$x_{i+1} = \frac{R_{i+1}}{R_i} \cdot [x_i \cos(\sigma_i \alpha_i) - y_i \sin(\sigma_i \alpha_i)] \tag{3.20}$$

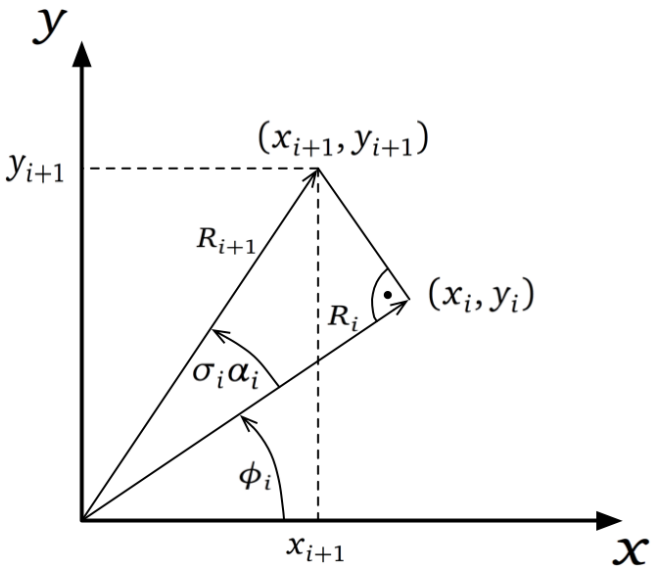


Abbildung 3.3.: Verzerrung der Vektorlänge bei jeder Iteration

Man erhält y_{i+1} analog und es ergibt sich:

$$y_{i+1} = \frac{R_{i+1}}{R_i} \cdot [y_i \cos \sigma_i \alpha_i + x_i \sin(\sigma_i \alpha_i)] \tag{3.21}$$

Der Vorfaktor wird definiert zu $\frac{R_{i+1}}{R_i} := k_i$ und man erhält die Iterationsvorschriften:

$$x_{i+1} = k_i \cdot [x_i \cos(\sigma_i \alpha_i) - y_i \sin(\sigma_i \alpha_i)] \tag{3.22}$$
$$y_{i+1} = k_i \cdot [y_i \cos(\sigma_i \alpha_i) + x_i \sin(\sigma_i \alpha_i)] \tag{3.23}$$

Durch die Symmetrie des Kosinus ($\cos(\sigma_i \alpha_i) = \cos(\alpha_i)$) und die Spiegelsymmetrie der Sinusfunktion ($\sin(\sigma_i \alpha_i) = \sigma_i \sin(\alpha_i)$) lassen sich die Gleichugen vereinfachen:

$$x_{i+1} = k_i \cos(\alpha_i) \cdot [x_i - \sigma_i y_i \tan(\alpha_i)] \tag{3.24}$$
$$y_{i+1} = k_i \cos(\alpha_i) \cdot [y_i + \sigma_i x_i \tan(\alpha_i)] \tag{3.25}$$

Der Zusammenhang $\sin^2(\alpha_i) + \cos^2(\alpha_i) = 1$ lässt sich mit $\tan(\alpha_i) = \frac{\sin(\alpha_i)}{\cos(\alpha_i)}$ umstellen:

$$\cos(\alpha_i) = \frac{1}{\sqrt{1 + \tan^2(\alpha_i)}} \tag{3.26}$$

Damit lassen sich die Gleichungen 3.24 und 3.25 umformen.

$$x_{i+1} = k_i \cdot \frac{1}{\sqrt{1 + \tan^2(\alpha_i)}} \cdot [x_i - \sigma_i y_i \tan(\alpha_i)] \tag{3.27}$$
$$y_{i+1} = k_i \cdot \frac{1}{\sqrt{1 + \tan^2(\alpha_i)}} \cdot [y_i + \sigma_i x_i \tan(\alpha_i)] \tag{3.28}$$

Durch die Definition aus Gleichung 3.15 ergibt sich für den Tangens:

$$\tan(\alpha_i) = 2^{-i} \tag{3.29}$$

Die Iterationsvorschriften lassen sich weiter vereinfachen:

$$x_{i+1} = k_i \cdot \frac{1}{\sqrt{1+2^{-2i}}} \cdot [x_i - \sigma_i y_i 2^{-i}] \tag{3.30}$$

$$y_{i+1} = k_i \cdot \frac{1}{\sqrt{1+2^{-2i}}} \cdot [y_i + \sigma_i x_i 2^{-i}] \tag{3.31}$$

Der Skalierungsfaktor k_i ergibt sich aus Abbildung 3.3 ($\cos(\alpha_i) = \frac{R_i}{R_{i+1}}$):

$$k_i = \frac{1}{\cos(\alpha_i)} = \sqrt{1+2^{-2i}} \tag{3.32}$$

Dann ergeben sich aus den Gleichungen 3.30 und 3.31 die Iterationsvorschriften:

$$x_{i+1} = x_i - \sigma_i y_i 2^{-i} \tag{3.33}$$

$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{3.34}$$

Der Wert x_i entspricht dem Sinuswert nach der i-ten Iteration, y_i dem Kosinus. Die Iterationsvorschriften benötigen keine Multiplikation mehr, sondern lediglich Schiebeoperationen und Additionen.

Während jeder Iteration muss die Drehrichtung σ_i bestimmt werden. Deshalb wird eine zusätzliche Variable eingeführt, die ebenfalls pro Iteration berechnet wird:

$$z_{i+1} := z_i - \sigma_i \alpha_i \tag{3.35}$$

Die Variable z_i gibt den aktuellen Winkel an. Der gesuchte Anfangswinkel wird mit $z_0 = \lambda$ definiert. Wenn z_i negativ ist, soll der nächste Teilwinkel addiert werden, bei positivem z_i subtrahiert. Weil in den angegeben Gleichungen σ_i subtrahiert wird, gilt:

$$\sigma_i = \begin{cases} -1, & \text{wenn } z_i < 0 \\ 1, & \text{wenn } z_i \geq 0 \end{cases} \tag{3.36}$$

In der Definition von z_{i+1} werden die Konstanten α_i verwendet. Diese Konstanten können vorab berechnet werden und verursachen so keinen weiteren Rechenaufwand.

Nach der Durchführung von n Iterationen müssen die Ergebnisse x_{n-1} und y_{n-1} aufgrund der fortlaufenden Betragsskalierung mit dem folgenden Faktor zurück auf den Einheitskreis normiert werden:

$$K = \prod_{i=0}^{n-1} k_i = \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}} \tag{3.37}$$

Bei einer Grenzwertbetrachtung von K mit $n \to \infty$ läuft der Skalierungsfaktor gegen $K \approx 1,64676$. Dies entspricht einer Multiplikation mit $K^{-1} \approx 0,607253$. Der Skalierungsfaktor ist in dieser Implementierung zur Vereinfachung für alle Iterationsanzahlen gleich und wird durch eine angepasste Fließkommaarithmetik mithilfe von acht Shift Operationen und Additionen berechnet. Dies entspricht in etwa der Genauigkeit einer 16-bit Fließkommaarithmetik.

Mit der hier vorgestellten Version des CORDIC-Algorithmus lässt sich die Sinus- und Kosinusfunktion approximieren und wird als *Rotation* bezeichnet. Charakteristisch daran ist die Konvergenz des Winkels z_i gegen Null. Des Weiteren gibt es noch das *Vectoring*. Dort lässt sich die Phase und der Betrag eines Vektors bestimmen. Die Variable y_i konvergiert dabei gegen Null. Außerdem lässt sich das CORDIC-Verfahren verallgemeinern, sodass neben den trigonometrischen weitere mathematische Funktionen realisierbar sind. Auf das Prinzip wird hier nicht näher eingegangen, weil es für diese DDS Implementierung nicht benötigt wird.

Das CORDIC-Verfahren erwartet drei Eingabewerte bzw. Anfangswerte. Je nachdem, wie die Anfangswerte gewählt werden, erhält man mit den unterschiedlichen Betriebsarten verschiedene mathematische Funktionen. Für eine Ausgabe von Sinus- und Kosinuswerten werden die Anfangswerte im Rotationsmodus mit $x_0 = 1$, $y_0 = 0$ und $z_0 = \lambda$ gewählt. Mit diesen Parametern erhält man $x_n = \sin(z_0)$, $y_n = \cos(z_0)$ und $z_n = 0$. Zur Vermeidung aufwändiger Fließkommaarithmetik wird der Wert x_0 auf die geforderte Bitbreite skaliert ($2^{WIDTH-1} - 1$) und man erhält als Ergebnis $x_n = x_0 \cdot \sin(z_0)$ und $y_n = x_0 \cdot \cos(z_0)$. [Hahn91]

Neben dem CORDIC-Algorithmus gibt es weitere Approximationsmethoden, wie z.B. die Taylor-Reihe. Der CORDIC-Algorithmus verbraucht weniger Ressourcen im Vergleich zu Polynomapproximationen und vor allem keine Multiplikationen, wodurch er der Favorit bei einer Implementierung ist. [GaDS11] [Riss04]

Die Iterationen des CORDIC-Algorithmus können in Hardware unterschiedlich implementiert werden. Beim Zeitmultiplexen werden alle Iterationen auf den selben Ressourcen berechnet. Dabei wird zu jedem Takt eine Iteration ausgeführt. Nachdem alle Iterationen durchlaufen sind, liegt das Ergebnis eines Eingangswertes vor. Diese Methode ist ressourcenschonend, benötigt allerdings so viele Taktzyklen, wie Iterationen vorhanden sind, bis ein Ergebnis vorliegt.

Alternativ können die Iterationen in Hardware abgerollt werden. Dabei ist jedoch nur eine niedrige Taktfrequenz möglich, weil die Recheneinheit eine längere Zeit benötigt, bis alle Iterationsstufen durchlaufen sind. Durch das Einfügen von Registern zwischen den Iterationsstufen entsteht eine Pipeline, was eine höhere Taktfrequenz zulässt. Diese Methode liefert das erste Ergebnis nach einer von den Iterationen abhängigen Anzahl an Taktzyklen. Dafür kann die Pipeline zu jedem Takt einen Wert aufnehmen. Nachdem das erste Ergebnis vorliegt, liefert die Pipeline zu jedem Takt ein Ergebnis. Die Latenz der Pipeline muss bei der anschließenden Amplitudenskalierung und Kanalauswahl berücksichtigt werden. Durch die Pipeline lässt sich eine DDS ohne LUT implementieren, die in jedem Takt eine Amplitude berechnen kann. Allerdings benötigt diese Variante mehr Ressourcen als die gezeitmultiplexte Implementierung.

Die gezeitmultiplexte Methodik sollte bei einer CORDIC-Initialisierung der LUT verwendet werden, um den Ressourcenverbrauch zu minimieren. Die Pipeline-Variante ist bei einer Amplitudenberechnung zur Laufzeit sinnvoll.

3.3.3 Lineare Interpolation

Eine Reduktion der in einer LUT abgelegten Stützstellen verringert zwar den Speicherbedarf und dessen Initialisierungszeit, verstärkt aber gleichzeitig die Quantisierung des erzeugten Signals. Um dem entgegenzuwirken lässt sich zwischen den Stützstellen linear interpolieren. Eine lineare Interpolation ist bei stetigen Funktionen sinnvoll, weil keine Funktionssprünge vorliegen. Sie lässt sich damit bei einer Sinusfunktion anwenden. Nicht förderlich wäre eine lineare Interpolation zum Beispiel bei Rechtecksignalen, wie der Pulsweitenmodulation (PWM).

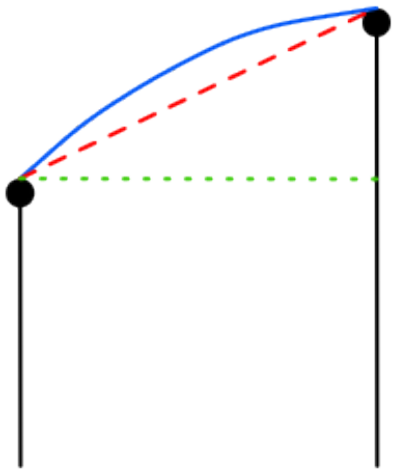


Abbildung 3.4.: Lineare Interpolation zwischen zwei Stützstellen

Abbildung 3.4 zeigt eine lineare Interpolation (rot) zwischen zwei aufeinanderfolgenden Stützstellen (schwarz) einer Sinusfunktion (blau). Ohne Interpolation (grün) werden nur die zwei Stützstellen als Amplitude verwendet. Der interpolierte Verlauf kann den idealen Verlauf hingegen deutlich besser annähern.

In dieser DDS Implementierung wird der Phasenakkumulator in zwei Anteile aufgeteilt, wie in Abbildung 3.5 gezeigt.

Abbildung 3.5.: Phasen-Akkumulator Aufteilung

Die höchst signifikanten Bits werden als Adresse (*addr*) für den RAM verwendet. Die restlichen Bits haben dabei keinen direkten Einfluss auf die Adresse. Sie geben stattdessen den *Interpolationsfaktor* an, mit dem zwischen den Stützstellen von *addr1* und

$addr1 + 1 = addr2$ interpoliert wird. Dieser Faktor repräsentiert die Tendenz, an welcher Stelle sich gerade die Amplitude zwischen den Stützstellen befindet. Hierfür muss ein Amplituden-Generator verwendet werden, der zwei Werte pro Takt liefern kann. Bei einer LUT-Realisierung setzt dies die Verwendung von Dual-Port RAMs voraus. Unter Verwendung des CORDIC-Algorithmus ohne RAM muss dieser zweimal instanziiert werden. Bei der Pipeline-Variante müssen die Interpolationsfaktoren über die Pipelinestufen mitgeführt werden, da die Interpolation erst nach Ablauf der Pipelinelatenz erfolgen kann.

Konkret lässt sich die lineare Interpolation mit Gleichung 3.38 realisieren.

$$to\_DAC = amp1 + \frac{amp2 - amp1}{2^{\overline{Interpolationsfaktor}}} \cdot Interpolationsfaktor \tag{3.38}$$

amp1, *amp2* sind die Amplitudenwerte zu *addr1* und *addr2*. Mit $\overline{Interpolationsfaktor}$ ist die Bitbreite des *Interpolationsfaktor*s gemeint.

3.4 Ansteuerung des Digital-Analog-Wandlers

Zur Umwandlung der durch die DDS berechneten digitalen Signale in analoge Spannungspegel wurde der Digital-Analog-Konverter DAC8728 [Texa09] von Texas Instruments verwendet. Das zugehörige Entwicklungsboard DAC8728EVM [Texa10] besitzt etliche Einstellungsmöglichkeiten per Jumper und Schnittstellen zur Anbindung externer Systeme.

Der DAC verfügt über acht unabhängige Kanäle mit einer Auflösung von 16 Bit, benötigt aber eine 5 Bit breite Adresse für zusätzliche Konfigurationen. Weitere Steuersignale dienen dem Zurücksetzen des DAC sowie dem Starten der Konvertierungen. Das mit maximal 50 MHz getaktete parallele Interface erlaubt eine schnelle Ansteuerung der Kanäle. [Texa10]

Um einem TI spezifischen Mikrokontroller-Interface gerecht zu werden, fasst das *DAC8728EVM* verschiedene Steuerleitungen zusammen und trennt die Adress- und Datenübertragung durch zwei Latches (*U10*, *U11*) in zwei Schritte. Dies erhöht die Zugriffszeit und erfordert eine vom eigentlichen *DAC8728*-Controller unterschiedliche Ansteuerungslogik.

Aus diesen Gründen wurde das Board so angepasst, dass die Latches und Demultiplexer umgangen werden. Das 16-Bit Latch wird über die $\overline{OE}$ Pins permanent aktiviert und alle Adressleitungen direkt an eine freie 20-Pin Buchse gelegt. Die Steuersignale wurden ebenfalls an diese Buchse gelegt und die Demultiplexer deaktiviert.

Die neue Ansteuerung benötigt zehn Leitungen, wovon aber fünf Leitungen die Adresse sind. Das Board benötigt nun zwei Leitungen mehr als bei der Lösung von TI, allerdings ist der Datentransfer auch zweimal schneller. Mit dem modifizierten Boards sind zwei Takte nötig, bis ein Wert in den DAC geschrieben wurde. Im ersten Takt werden die Adresse und die Daten auf die Busse gelegt. Im zweiten Takt wird die Steuerleitung $\overline{CS}$ deaktiviert, damit die Daten in den DAC geschrieben werden. Diese Daten werden mit Triggern des Steuersignals $\overline{LDAC}$ konvertiert.

Die Ansteuerung des modifizierten Boards ist identisch zur Ansteuerung des DACs ohne das Entwicklungsboard und wurde im Modul *DAC8728* implementiert. Der modifizierte Schaltplan ist im Anhang A.1 zu finden.

Der DAC wird so angesteuert, dass er Daten, die er pro Kanal entgegennimmt, sofort in einen analogen Wert konvertiert. So ist sichergestellt, dass keine ungewollte Phasenverschiebung zwischen den Kanälen vorliegt. Die Alternative zu diesem Vorgehen wäre, alle Werte der Kanäle in den DAC zu schreiben und dann die Konvertierung von allen Kanälen zu starten. Dies würde jedoch eine Phasenverschiebung mit sich führen, weil die Werte zum Konvertierungszeitpunkt nicht mehr aktuell wären.

3.5 Tiefpassfilter

Bei digital erzeugten Signalen hat das Ausgangssignal eine stufige Form, weil im digitalen Bereich mit diskreten Werten gearbeitet wird. Abbildung 3.6 zeigt ein per DDS generiertes Sinussignal bei einer LUT-Tiefe von 16.

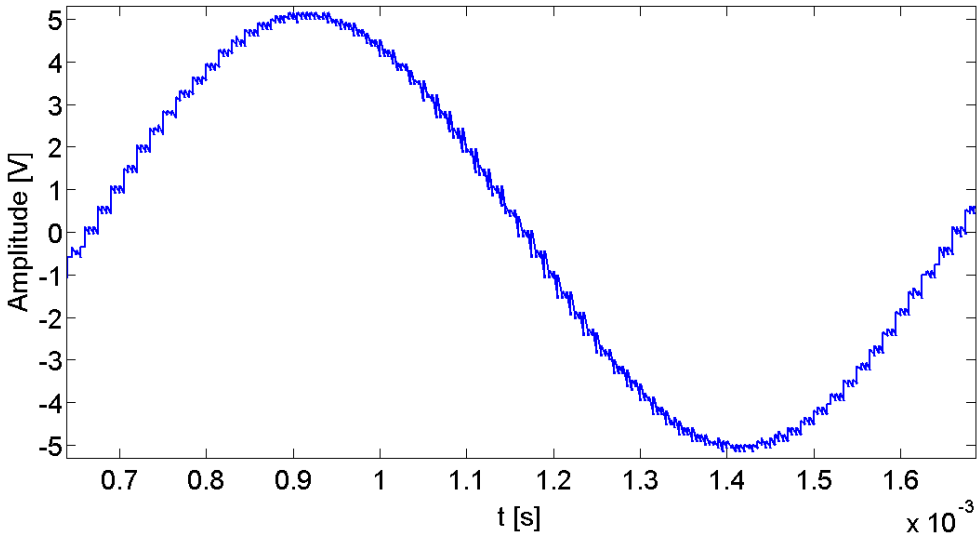


Abbildung 3.6.: Sinussignal bei 1 kHZ mit überlagertem Rechtecksignal

Das abgebildete Signal zeigt ein Sinussignal mit einem überlagerten Rechtecksignal. Durch die Rechteck-Anteile ergibt sich ein breiteres Frequenzspektrum. Die Frequenz des Rechtecksignals lässt sich mit Gleichung 3.39 bestimmen.

$$f_{Rechteck} = AnzahlStützstellen \cdot f_{Sinus} = 2^{DEPTH\_SAMPLES} \cdot 4 \tag{3.39}$$

Die Frequenz des überlagerten Rechtecksignals aus Abbildung 3.6 liegt demnach bei 64 kHz. Das Sinussignal kann außerdem Oberwellen durch Ungenauigkeiten in der digitalen Amplitudenerzeugung und der Digital-Analog-Umwandlung aufweisen. Die Frequenzen der Oberwellen eines Sinussignals befinden sich bei dem Vielfachen der Grundfrequenz. Ein dem DAC nachgeschalteter Tiefpassfilter mit einer Grenzfrequenz von $f_g = f_{Grundwelle}$ kann sowohl die Oberwellen des Sinussignals, als auch die Rechteckanteile dämpfen.

Es gibt aktive und passive Tiefpassfilter. Aktive Tiefpassfilter haben mindestens ein aktives Bauteil in der Schaltung, z.B. einen Operationsverstärker. Passive dagegen greifen nur auf Widerstände, Kondensatoren und Induktivitäten zurück.

Ein Tiefpassfilter besitzt pro Ordnungszahl einen Energiespeicher. Je höher die Ordnungszahl eines Filters ist, desto steiler ist dessen Dämpfung. Zur Realisierung eines TPF höherer Ordnung können Filter erster und/oder zweiter Ordnung hintereinander geschaltet. Dabei müssen die einzelnen Filter unbelastet sein. Das heißt, nachgeschaltete Filter haben keine Rückwirkung auf ihre Vorgänger. Dies wird mithilfe eines aktiven Bauteils erreicht. So kann z.B. ein Operationsverstärker oder Transistor verwendet werden, der zwischen den hintereinander geschalteten Filtern liegt. [Schm11a]

Bei einem Tiefpassfilter n-ter Ordnung beträgt die Dämpfung $n \cdot 20$ dB pro Dekade.

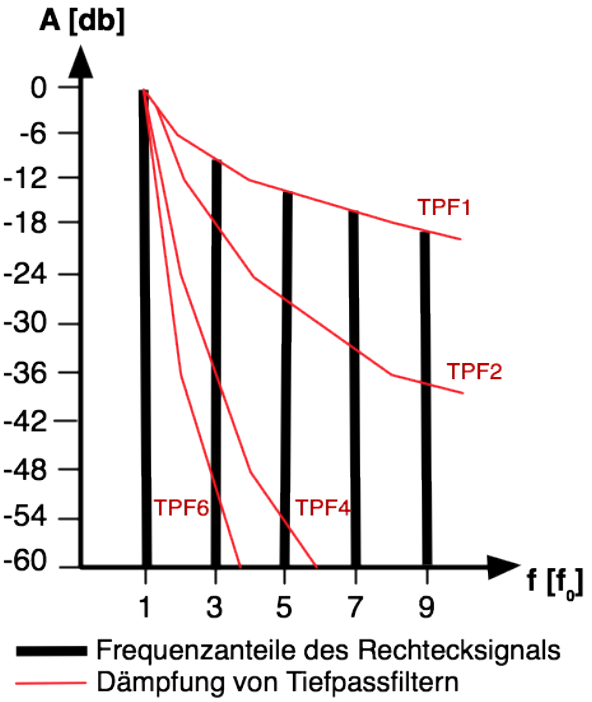


Abbildung 3.7.: Frequenzspektrum eines Rechtecksignals mit Dämpfungskennlinien von unterschiedlichen TPFs

Abbildung 3.7 zeigt den Verlauf der Dämpfung von Tiefpassfiltern unterschiedlicher Ordnung im Frequenzspektrum eines idealen Rechtecksignals mit Grundfrequenz f_0.

Der Tiefpass erster Ordnung (*TPF1*) dämpft die Oberwellen nur minimal bzw. gar nicht. Ein Tiefpassfilter sechster Ordnung dagegen besitzt eine sehr hohe Flankensteilheit und filtert die Grundwelle des gegebenen Signals besser. Er lässt in dem Beispiel von Abbildung 3.7 die Grundwelle mit einer stark gedämpften ersten Oberwelle und sehr stark gedämpften höheren Oberwellen passieren.

Ein Tiefpassfilter erster Ordnung benötigt genau einen Energiespeicher und muss daher entweder eine Spule oder einen Kondensator beinhalten. In Abbildung 3.8 ist ein solcher RC-Tiefpassfilter abgebildet.

Auf der linken Seite der Schaltung befindet sich die Eingangsspannung (U_e), die gefiltert werden soll. Durch den Kondensator, der gegen Masse geschaltet ist, erreichen

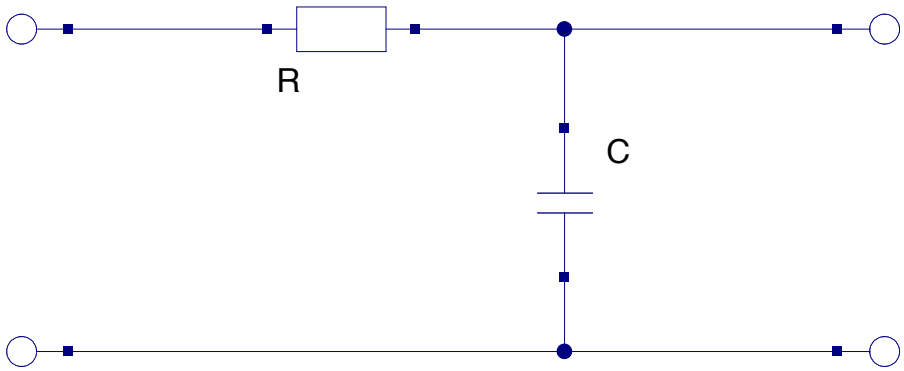


Abbildung 3.8.: RC-Tiefpassfilter erster Ordnung

hohe Frequenzen nicht den Ausgang rechts (U_a). Ein Kondensator ist für hohe Frequenzen niederohmig, weshalb Signale mit hohen Frequenzen über den Kondensator C leicht die Masse erreichen. Diese Frequenzanteile sind nur noch abgeschwächt im Ausgangssignal vorhanden.

Dieses Verhalten lässt sich an der Impedanzgleichung zur Übertragungsstrecke des RC-Tiefpassfilters aus Abbildung 3.8 erkennen. U_a fällt über dem Kondensator C ab, U_e über C und R. Es ergibt sich:

$$\underline{Z}(j\omega) = \frac{\underline{U}_a(j\omega)}{\underline{U}_e(j\omega)} = \frac{\underline{Z}_C}{R + \underline{Z}_C} = \frac{\frac{1}{j\omega C}}{R + \frac{1}{j\omega C}} = \frac{1}{1 + j\omega RC} \tag{3.40}$$

Die Gleichung wird normiert mithilfe der komplexen Kreisfrequenz $s = j\omega$:

$$\underline{Z}(s) = \frac{1}{1 + sRC} \tag{3.41}$$

Wir betrachten für $\underline{U}_a(s) = \underline{U}_e(s) \cdot \underline{Z}(s) = \underline{U}_e(s) \cdot \frac{1}{1+sRC}$ in Gleichung 3.42 und 3.43 die Grenzwerte für die komplexe Kreisfrequenz s.

$$\lim_{s\to 0} \underline{U}_a = \lim_{s\to 0} \underline{U}_e \cdot \frac{1}{1 + sRC} = \underline{U}_e \tag{3.42}$$

$$\lim_{s\to \infty} \underline{U}_a = \lim_{s\to \infty} \underline{U}_e \cdot \frac{1}{1 + sRC} = 0 \tag{3.43}$$

Bei kleinen Frequenzen bzw. bei Gleichspannung ($f = 0$ Hz) wird die Ausgangsspannung $\underline{U}_a$ annähernd gleich $\underline{U}_e$. Signalbestandteile mit hoher Frequenz werden hingegen nicht vom Eingang auf den Ausgang übertragen..

Durch die Definition der Grenzfrequenz ($\frac{U_a}{U_e} = \frac{1}{\sqrt{2}}$) ergibt sich aus Gleichung 3.40:

$$\frac{1}{\sqrt{2}} = \left| \frac{1}{1 + j\omega_g RC} \right| \tag{3.44}$$

$$\frac{1}{\sqrt{2}} = \frac{1}{\sqrt{1 + (\omega_g RC)^2}} \tag{3.45}$$

$$\frac{1}{2} = \frac{1}{1 + \omega_g^2 R^2 C^2} \tag{3.46}$$

$$2 = 1 + \omega_g^2 R^2 C^2 \tag{3.47}$$

Ersetzen der Kreisfrequenz durch die Frequenz ($\omega_g = 2\pi f_g$) und Auflösen nach f_g ergibt schließlich die Grenzfrequenz aus Gleichung 3.48. [Sche08] [Schm11a] [Umni11]

$$f_g = \frac{1}{2\pi RC} \tag{3.48}$$

Bild 3.9 zeigt einen Tiefpassfilter zweiter Ordnung.

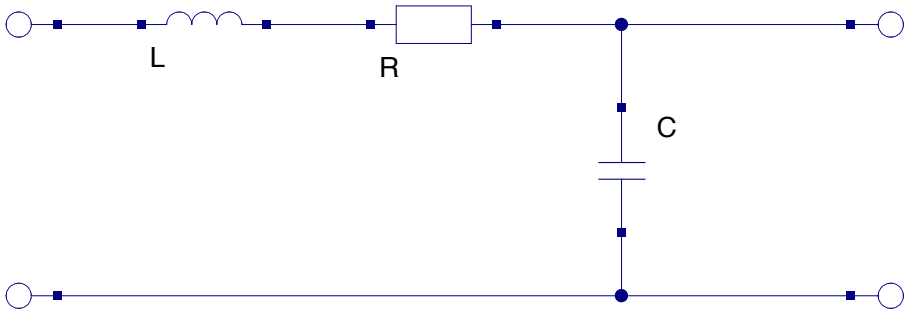


Abbildung 3.9.: LC-Tiefpassfilter zweiter Ordnung

Die Schaltung besitzt mit der Spule L einen weiteren Energiespeicher. Eine Spule ist für hohe Frequenzen hochohmig und lässt deshalb nur kleine Frequenzen passieren. Gleichung 3.49 zeigt die Impedanzgleichung eines LC-Tiefpasses zweiter Ordnung.

$$\underline{Z}(j\omega) = \frac{\underline{U}_a(j\omega)}{\underline{U}_e(j\omega)} = \frac{\frac{1}{j\omega C}}{j\omega L + R + \frac{1}{j\omega C}} = \frac{1}{1 + j\omega RC + (j\omega)^2 LC} \tag{3.49}$$

Die Gleichung wird wieder mit $s = j\omega$ normiert und ist in Gleichung 3.50 dargestellt.

$$\underline{Z}(s) = \frac{1}{1 + sRC + s^2 LC} \tag{3.50}$$

Eine Grenzwertbetrachung mit $s \to 0$ und $s \to \infty$ zeigt, dass hohe Frequenzen blockiert und niedrige Frequenzen durchgelassen werden.

Die Grenzfrequenz lässt sich in diesem Fall nicht mehr so einfach bestimmen. Zur Auslegung komplexer Filter sei daher auf [Schm11a] verwiesen.

Die charakteristischen Eigenschaften passiver Bauteile können vom Nennwert innerhalb einer vom Hersteller spezifizierten Toleranz abweichen. Wie sich die Toleranz auf die Grenzfrequenz bei einem Tiefpassfilter erster Ordnung auswirkt, zeigt Gleichung 3.51. Dabei wird auf Gleichung 3.48 zurückgegriffen.

$$f_g = \frac{1}{2\pi(R+\Delta R)(C+\Delta C)} \tag{3.51}$$

ΔR und ΔC beschreibt die Abweichung zum angegeben Fertigungswert. Bei einer prozentualen Toleranz ergibt sich:

$$f_g = \frac{1}{2\pi(R+RT_R)(C+CT_C)} \tag{3.52}$$

Bei derselben Toleranz $T = T_R = T_C$ resultiert die Grenzfrequenz zu:

$$f_g = \frac{1}{2\pi RC(1+T)^2} \tag{3.53}$$

Die relative Abweichung von der idealen Grenzfrequenz beläuft sich auf $1 - \frac{1}{(1+T)^2}$.

Es sollten daher Bauteile mit einer kleinen Toleranz verwendet werden, um die bestmöglichen Ergebnisse zu erhalten.

3.6 Schnittstellen

Abschließend folgt eine Übersicht über die Parameter und Ports der Module *DDS* und *DAC8728*. Tabelle 3.1 zeigt das Interface der DDS. Alle Signale mit einem ”N” am Ende ihres Namens sind active-low.

Parameter	**In/Out**	**Beschreibung**
FREQ_DIGITAL	In	Taktfrequenz des *clk* Signals
FREQ_OUT_LOW	In	Untere Schranke der Ausgabefrequenz
FREQ_OUT_HIGH	In	Obere Schranke der Ausgabefrequenz (wegen Rundungen kann die Ausgabefrequenz größer sein)
FREQ_OUT_RESOLUTION	In	Maximale Frequenzauflösung in Hz (wegen Rundungen ggf. kleiner)
CHANNEL_AMOUNT	In	Anzahl aktiver Kanäle
DEPTH_SAMPLES	In	Anzahl Samples bzw. Stützstellen = $2^{DEPTH\_SAMPLES} \cdot 4$; Bitbreite der Adressleitung zur LUT oder zum CORDIC-Algorithmus
WIDTH_SAMPLES	In	Bitbreite der Stützstellen

WIDTH_AMP_SCALE	In	Bitbreite der Amplitudenskalierung
WIDTH_DAC	In	Bitbreite des parallelen Bus des DACs
FLASH_DELAY	In	Taktzyklen, die der Flash für die Ausgabe eines Datensatzes benötigt
STEPS_DAC	In	Taktzyklen, die der DAC für die Konvertierung eines Wertes benötigt
CORDIC_ITERATIONS	In	Anzahl der Iterationen, die der CORDIC-Algorithmus verwenden soll
RAM_CORDIC_N[0:0]	In	Amplitudengenerierung per LUT (1) oder CORDIC-Algorithmus (0)
FLASH_CORDIC_N[0:0]	In	Initialisierung der LUT per Flash-Speicher (1) oder CORDIC (0)
INTERPOLATION_BETWEEN_SAMPLES[0:0]	In	Aktivierung der Interpolation zwischen benachbarten Stützstellen
CORDIC_PIPELINED[0:0]	In	Schaltet die Iterationsstufen des CORDIC-Algorithmus hintereinander, wodurch bei jeder Taktflanke ein Ausgabewert berechnet wird
CORDIC_CONSTANTS[*CORDIC_ITERATIONS*·*WIDTH_SAMPLES*:0]	In	Konstanten für die angegebene Anzahl an Iterationsschritten des CORDIC-Algorithmus. Muss bei *WIDTH_SAMPLES* $\neq$ 16 oder *CORDIC_CONSTANTS* $>$ 15 angepasst werden. LSB für Iteration Null.
Port	**In/Out**	**Beschreibung**
clk	In	Taktsignal (steigende Flanke)
resetN	In	Setzt die DDS zurück (asynchroner Reset)
flashData[31:0]	In	Datenbus eines Flashs (nicht benutzt, wenn der CORDIC-Algorithmus verwendet wird)
sum[*WIDTH_ADD*·*CHANNEL_AMOUNT*-1:0]	In	Individueller Sprungwert
phase_shift[*WIDTH_ACC*·*CHANNEL_AMOUNT*-1:0]	In	Phasenverschiebung
amp_scale[*WIDTH_AMP_SCALE* · *CHANNEL_AMOUNT*-1:0]	In	Amplitudenskalierung
rdy	Out	Zeigt an, ob die Initialisierung abgeschlossen ist
flashHiCeN	Out	Chip Enable für den ersten Flash-Baustein
flashLoCeN	Out	Chip Enable für den zweiten Flash-Baustein
flashWeN	Out	Write Enable für den Flash
flashOeN	Out	Output Enable für den Flash
flashRstN	Out	Reset für den Flash
flashAddr[23:0]	Out	Adressbus für den Flash

to_DAC[*WIDTH_DAC*-1:0]	Out	Der vom DAC zu konvertierende Wert
channel[$\lceil \log_2(CHANNEL\_AMOUNT + 1) \rceil$-1:0]	Out	Kanal, für den der Wert *to_DAC* bestimmt ist

Tabelle 3.1.: Interface der DDS

Die Eingangsports *sum*, *phase_shift* und *amp_scale* beinhalten die Werte für alle aktiven Kanäle, wobei der Kanal 0 durch die am wenigsten signifikanten Bits dieser Ports gesteuert wird. Die Signalbreiten der Sprungwerte (*WIDTH_ADD*) und der Phasenverschiebungen (*WIDTH_ACC*) werden von der DDS anhand der Frequenzauflösung und der Taktfrequenz berechnet.

Das Interface für den *DAC8728* ist in Tabelle 3.2 dargestellt.

Port	**In/Out**	**Beschreibung**
clk	In	Taktsignal (steigende Flanke)
resetN	In	Setzt das Modul zurück (asynchroner Reset)
value[15:0]	In	Wert, der vom DAC konvertiert werden soll
channel[2:0]	In	Kanal, für den der Wert *value* bestimmt ist
DAC_parallelData[15:0]	Out	Daten, die über den parallelen Bus des DACs gesendet werden
DAC_addr[4:0]	Out	Adresse der Register, die im DAC beschrieben werden sollen
DAC_RNW	Out	Aktiviert das Lesen der DAC-internen Register
DAC_CLRN	Out	Setzt die analogen Ausgänge zurück
DAC_RSTN	Out	Setzt den DAC zurück (die DAC internen Register)
DAC_LDACN	Out	Konvertiert die digitalen Werte in eine Spannung
DAC_DC_CSN	Out	Aktiviert den DAC

Tabelle 3.2.: Interface des *DAC8728*

4 Analyse

Zur Implementierung der Direct Digital Synthesis wird ein FPGA-Board (FPGA - Field Programmable Gate Array) verwendet. Bei dem Development Board handelt es sich um das *M1AGL1000-DEV-KIT* [Acte11] von Actel, welches mit einem *M1AGL1000V2* [Acte09] arbeitet. Das Board verfügt über einen 48-MHZ Oszillator. Mittels PLL (Phase-Locked Loop) lassen sich auch andere Frequenzen verwenden. Außerdem besitzt es zwei Flash-Bausteine mit insgesamt 16 MByte und zwei SRAM-Blöcke mit 1 MByte. [Acte11]

Dieses Kapitel stellt die Ergebnisse vor, die mit den gewählten Komponenten und Verfahren erreicht wurden. Dabei werden verschiedene Implementierungsdetails hinsichtlich ihrer Auswirkungen auf die Signalgüte und den Ressourcenbedarf der DDS analysiert.

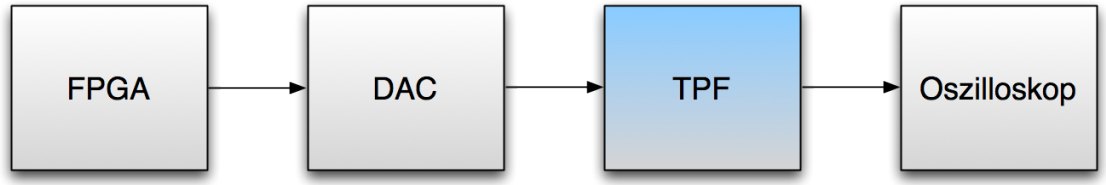


Abbildung 4.1.: Blockschaltbild des Messaufbaus

Abbildung 4.1 zeigt den Messaufbau. Das FPGA erzeugt die Werte, die an den DAC geschickt werden. Der Tiefpassfilter ist optional und wird nur bei der Analyse des TPFs verwendet. Das Oszilloskop ist mit dem Ausgang des DACs bzw. TPFs verbunden. Mit diesem werden die Messdaten gespeichert und auf einem PC ausgewertet. Die Abtastrate des Oszilloskops liegt bei Messungen von 150 kHz Signalen bei 10 MSa (Sa = Samples) und bei 1 kHz Signalen bei 8 MSa. Die LUT-Breite beträgt bei allen Messungen 16 Bit, wenn nicht anders angegeben.

Zur objektiven Beurteilung der Signalgüte wird der Klirrfaktor herangezogen, der das Verhältnis zwischen den Grund- und Oberwellen widerspiegelt. Da das ideale Sinussignal keinen Oberwellenanteil besitzt, stellt der Klirrfaktor ein gutes Maß für die Qualität des Signals dar. Er lässt sich mithilfe der Gleichung 4.1 berechnen. [Schl05] [Weiß05]

$$k = \sqrt{\frac{U_1^2 + U_2^2 + U_3^2 + \ldots}{U_0^2 + U_1^2 + U_2^2 + U_3^2 + \ldots}} \tag{4.1}$$

U_0: Effektivwert der Grundwelle

U_n: Effektivwert der n-ten Oberwelle

Das zur Berechnung des Klirrfaktors verwendete Matlab-Skript ist in A.2.1 zu finden.

Um die Genauigkeit des Messverfahrens zu bestimmen, wurden die Klirrfaktoren verschiedener DDS-Implementierungen (Amplitudengenerierung per LUT mit 16 bzw. 64 Einträgen) mit denen eines kommerziellen Signalgenerators [Agil11] verglichen.

	1 kHz	**150 kHz**
Signalgenerator	0,9715 %	0,3591%
DDS: LUT mit 16 Stützstellen	2,594 %	3,9316 %
DDS: LUT mit 64 Stützstellen	1,0593 %	3,914 %

Tabelle 4.1.: Vergleich zwischen kommerziellen Signalgenerator und DDS

Betrachten wir zunächst die Erzeugung eines 1 kHz Signals. Der Signalgenerator weist mit der hier verwendeten Messmethodik einen Klirrfaktor von knapp unter einem Prozent auf. Bei einer solch geringen Frequenz hängt die Qualität des per DDS generierten Signals, vor allem von der Anzahl der Stützstellen ab. Dies erkennt man deutlich an den Klirrfaktoren bei den unterschiedlichen DDS Konfigurationen in der Tabelle. Ein Sinussignal mit 64 Stützstellen in der LUT (dies ergibt nach Gleichung 3.11 eine gesamte Anzahl an Stützstellen von 256) erreicht bei der DDS schon einen Klirrfaktor, der recht nahe an der Qualität des Signalgenerators liegt.

Bei einer Ausgabefrequenz von 150 kHz erreicht der Signalgenerator wesentlich bessere Klirrfaktor-Werte als die DDS. Selbst mit 64 Stützstellen in der LUT liegt die DDS in der Qualität weit hinter dem Signalgenerator.

Die Ressourcen, die das FPGA für die getesteten Methoden benötigt, orientieren sich an den verwendeten Block-RAMs, CoreCells (Logikeinheiten) und der maximal möglichen Taktrate. Alle Angaben zum Ressourcenverbrauch sind Post-Synthese-Werte (mit Resource Sharing, ohne Retiming, Zielfrequenz von 25 MHz), welche mit *Synplify Pro E-2010.09A-1* von *Synopsys* ermittelt wurden.

Durch die regelmäßigen kleinen Time-Slots, die zur Berechnung der aktuellen Phase und Amplitude benötigt werden, ist kaum ein sinnvolles Power Management möglich. Die DDS kann nicht in StandBy geschaltet werden, weil die Zeit zwischen den Slots zu klein ist. Der in der Implementierung verwendete FPGA besitzt einen energiesparenden ”Flash Freeze” sleep-mode. Die Zeit, bis das FPGA alle Register und Ausgänge gesichert bzw. ausgelesen hat, beläuft sich auf etwa 1 μs. [EnLK11] Die DDS ermöglicht bei einer typischen Konfiguration mit einer FPGA Frequenz von 15 MHz und einem DAC, der zwei Taktzyklen pro Konvertierung benötigt, Time-Slots von $\frac{2}{15\ \text{MHz}} = 0,1\overline{3}\ \mu s$. Somit lässt sich keine Energie durch das vorübergehende Abschalten einsparen. Es lässt sich kein signifikanter Unterschied im Energieverbrauch messen, deshalb beinhaltet die Analyse keine weiteren Untersuchungen zum Energieverbrauch.

4.1 Controller-Frequenz

Die Qualität des Ausgangssignals hängt von der Frequenz ab, mit der die Recheneinheit betrieben wird.

Das Diagramm in Abbildung 4.2 zeigt den Klirrfaktor für unterschiedlich getaktete DDS-Realisierungen. Die LUT beinhaltete 16 Stützstellen und wurde mithilfe des

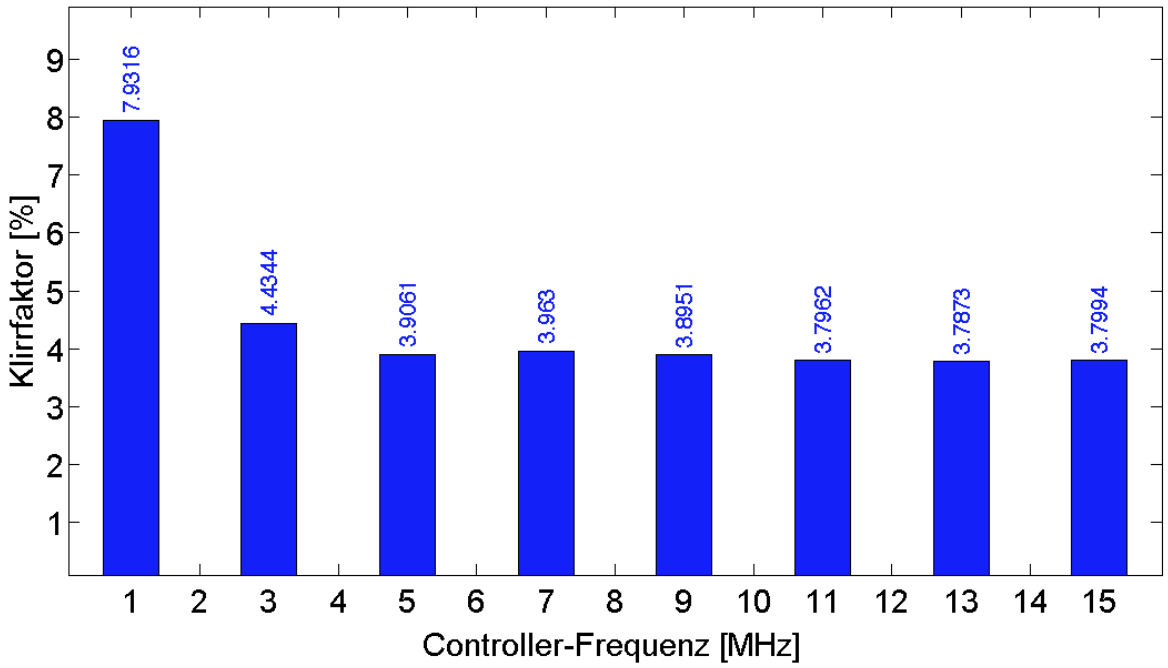


Abbildung 4.2.: Klirrfaktor in Bezug auf Controller-Frequenz

CORDIC-Algorithmus gefüllt. Die Messungen beziehen sich auf einen aktiven Kanal mit 150 kHz.

Wenn die Frequenz unterhalb von 5 MHz liegt, nimmt die Signalqualität ab. Frequenzen größer gleich 5 MHz zeigen keinen signifikanten Unterschied bei der Verwendung eines aktiven Kanals.

Bei einer niedrigen Controller-Frequenz werden nur wenige Stützstellen für die Erzeugung des Signals verwendet, wodurch sich dessen Güte verschlechtert. Wie viele Stützstellen tatsächlich in einem Ausgangssignal zu finden sind, wenn alle Ausgangskanäle dieselbe Frequenz aufweisen, kann mithilfe von Gleichung 4.2 berechnet werden.

$$samples = \frac{FREQ\_DIGITAL}{STEPS\_DAC \cdot CHANNEL\_AMOUNT \cdot f_{out}} \tag{4.2}$$

f_{out} ist die Ausgangsfrequenz, welche sich nach Gleichung 3.5 berechnen lässt.

Tabelle 4.2 zeigt für exemplarische Controller-Frequenzen die Anzahl der verwendeten Stützstellen pro Periode, die an den DAC geschickt werden. Diese theoretischen Werte beziehen sich ebenfalls auf einen aktiven Kanal mit 150 kHz.

Bei 1 MHz Controller-Frequenz werden nur $3,\overline{3}$ Stützstellen pro Periode verwendet. Das sind zu wenige, um ein Sinussignal ohne anschließende Tiefpassfilterung sauber darzustellen. Wieso die steigende Anzahl an Stützstellen, die an den DAC geschickt werden, den Klirrfaktor nicht beeinflusst, wird im Laufe der Arbeit erläutert.

Controller-Frequenz [MHz]	Anzahl verwendeter Stützstellen
1	$3,\overline{3}$
3	10
5	$16,\overline{6}$
9	30
13	$43,\overline{3}$

Tabelle 4.2.: Anzahl der verwendeten Stützstellen in Abhängigkeit von der Controller-Frequenz

4.2 Anzahl der erzeugten Ausgabekanäle

Bei der Verwendung von mehreren Kanälen wird die Ausgaberate des DACs gleichmäßig auf alle Kanäle verteilt. Mehr Kanäle bedeuten weniger Stützstellen, die pro Kanal ausgegeben werden. Die Signalgüte hängt somit von der Anzahl der aktiven Kanäle ab.

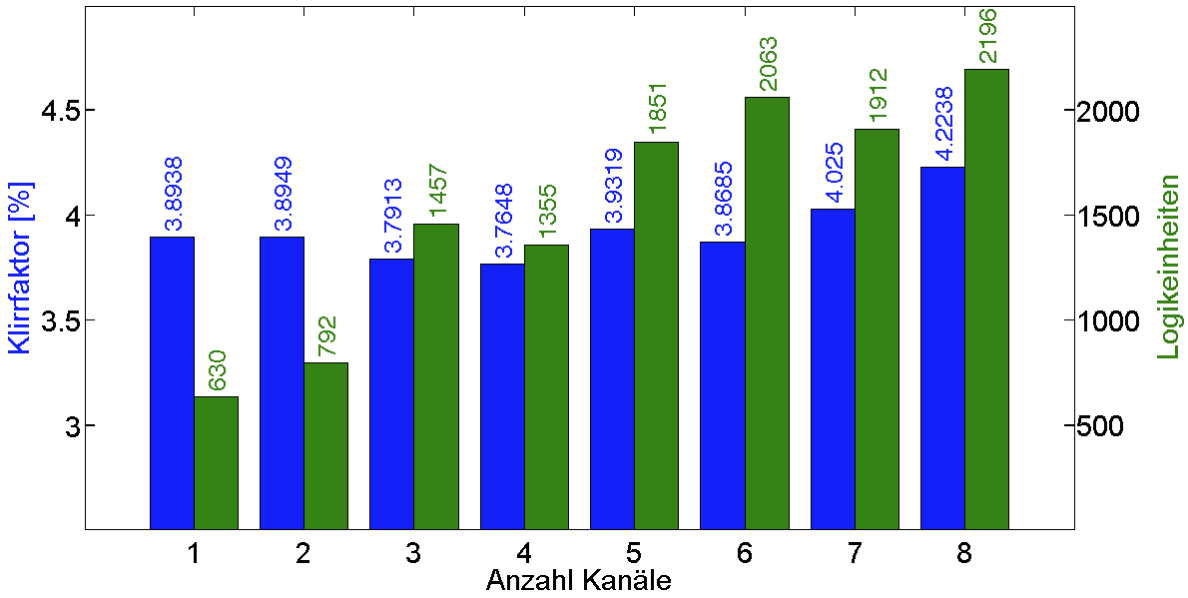


Abbildung 4.3.: Klirrfaktor in Bezug auf Anzahl der Kanäle

Wie sich die Anzahl der Kanäle auf den Klirrfaktor auswirkt, stellt Abbildung 4.3 dar. Das Diagramm zeigt den Klirrfaktor (blaue Balken) und die benötigten Logikeinheiten (grünen Balken) bei steigender Kanalanzahl. Alle Kanäle wurden auf 150 kHz eingestellt. Die mithilfe des externen Flash-Speichers gefüllte LUT beinhaltete 16 Stützstellen und die FPGA Frequenz betrug 15 MHz.

Der Klirrfaktor schwankt um ca 3,9 %. Eine Qualitätsverschlechterung ist erst bei acht aktiven Kanälen zu sehen, wo der Klirrfaktor sein Maximum erreicht.

Mehr Kanäle bedeuten einen höheren Verbrauch an Logikeinheiten. Bei einem Kanal ist der Ressourcenverbrauch am geringsten. Bei steigender Kanalanzahl erhöht sich der Ressourcenverbrauch sprunghaft. Der Sprung von einem auf zwei Kanäle lässt sich

durch die benötigten Multiplexer erklären. Bei drei und vier Kanälen wird die benötigte Bitbreite der Multiplexer erhöht, was in einem höheren Ressourcenverbrauch endet. Ab fünf Kanälen wird die Bitbreite erneut erhöht und es folgt ein um 1900 CoreCells schwankender Verlauf. Auf dem verwendeten FPGA sind 24576 CoreCells vorhanden.

4.3 RAM Initialisierungsalternativen

Wie bereits in Kapitel 3.3.1 vorgestellt, lässt sich eine Look-Up Table mit verschiedenen Methoden befüllen.

Bei einer Initialisierung aus einem externen Flash-Baustein müssen die Einträge auf einem PC (oder einer anderen Recheneinheit) vorberechnet und anschließend an die Bitbreite der LUT angepasst werden. Die Genauigkeit der Werte wird vor allem durch die Bitbreite der LUT begrenzt.

Wenn der CORDIC-Algorithmus zur LUT-Initialisierung verwendet wird, werden die Werte zur Laufzeit approximiert. Bei 15 CORDIC-Iterationene beträgt der mittlere quadratische Fehler 0,01685 %. Bei dieser geringen Abweichung haben die LUT-Initialisierungsmethoden keine signifikante Auswirkung auf die Signalqualität. Messungen haben bei einer Initialisierung mittels CORDIC-Algorithmus (15 Iterationen) einen Klirrfaktor von 3,9032 % ergeben, bei einer Initialisierung aus dem Flash-Speicher 3,8938 %. Wenn die Iterationsanzahl des CORDIC-Algorithmus verringert wird, ändern sich die generierten Werte. Wie sich dies auf die Signalgüte ausübt, wird in Kapitel 4.7 behandelt.

Die zur Initialisierung benötigte Zeit hängt linear von der LUT-Tiefe ab. Sie ist in den meisten Anwendungsfällen unkritisch. Wichtiger ist der Ressourcenverbrauch der beiden Initialisierungsmöglichkeiten. Ein Controller benötigt für die Ansteuerung des Flash-Speichers kaum Ressourcen. Es werden vor allem Leitungen zum Flash-Baustein verwendet, was IO Pins verbraucht. Anders sieht das bei dem CORDIC-Algorithmus aus. Bei diesem werden mehrere Register und vor allem Rechts-Shifter benötigt. Der CORDIC-Algorithmus erhöht den Ressourcenverbrauch, wie die nachfolgenden Werte zeigen.

Bei einem aktiven Kanal und einer Ausgangsfrequenz von 150 kHz benötigt eine Implementierung mit Flash-Initialisierung 630 Core Cells, etwa 2,6 % Auslastung des FPGAs. Eine vom Synthese-Tool maximal erreichbare, geschätzte Frequenz liegt bei 20,6 MHz. Bei denselben Konfigurationen werden mit einer CORDIC-Initialisierung 1736 Core Cells (7,1 %) benötigt. Die geschätzte Maximal-Frequenz verringert sich auf 18,4 MHz. Beide Varianten benötigen einen BRAM. Die Implementierung mittels CORDIC-Algorithmus benötigt wesentlich mehr Logikeinheiten und somit Platz auf dem FPGA als die Flash-Initialisierung.

Selbst bei der Initialisierung lässt sich der CORDIC-Algorithmus mit einer Pipeline verwenden. Das spart zwar Initialisierungszeit, steht aber in keinem Verhältnis zu den verbrauchten Ressourcen. Für eine Pipeline Konfiguration werden 3747 CoreCells (15,2 %) bei 15 Iterationen und einem aktiven Kanal benötigt. Die maximal geschätzte Frequenz liegt nur noch bei 5,8 MHz. Die Pipeline ist nur sinnvoll, wenn kein BRAM

verwendet werden soll.

Ein weiterer Unterschied zwischen den beiden Initialisierungsmethoden ist deren Flexibilität, was sich gerade in einer Erprobungsphase bemerkbar macht. Der CORDIC-Algorithmus ist durch die Anwendung zur Laufzeit in dieser Hinsicht sehr flexibel und praktisch, denn es müssen keine Werte von einem externen Medium oder von einer externen Schnittstelle zur Verfügung gestellt werden. Bei einer Flash-Initialisierung muss der Flash-Speicher bei einer Änderung der LUT-Tiefe erneut beschrieben werden.

Die Analyse zeigt, dass beide Verfahren Vor- und Nachteile haben. Wenn auf der Recheneinheit nicht viel Platz verfügbar und ein Flash-Speicher vorhanden ist, sollte der Flash für eine endgültige Implementierung verwendet werden.

4.4 Auswirkung der LUT-Tiefe

Die Anzahl der Stützstellen in der Look-Up Table hat direkten Einfluss auf die Qualität des Signals.

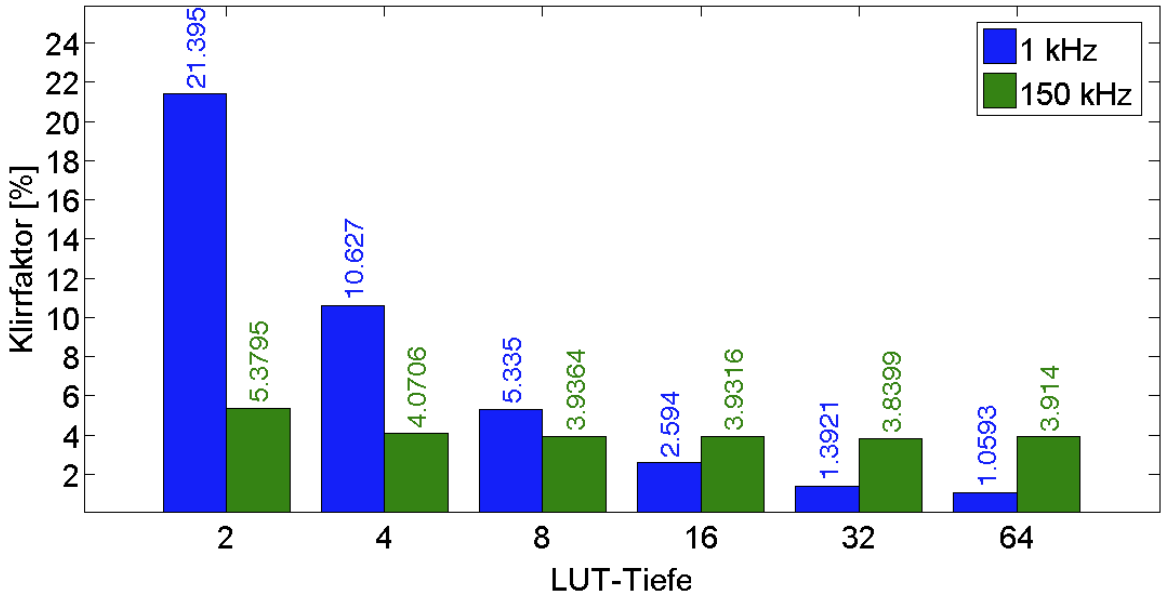


Abbildung 4.4.: Auswirkung der LUT Größe auf den Klirrfaktor

Abbildung 4.4 zeigt den Verlauf des Klirrfaktors bei einer unterschiedlichen LUT-Tiefe. Die x-Achse gibt die Tiefe der LUT an. Die daraus generierbaren Stützstellen ergeben sich nach Gleichung 3.11. Das Diagramm zeigt den Verlauf bei einem 1 kHz und bei einem 150 kHz Signal. Beide Signale wurden mit einem aktiven Kanal und bei einer FPGA Frequenz von 15 MHz generiert.

Der Klirrfaktor des 1 kHz Signals zeigt einen annähernd linearen Verlauf. Es wird deutlich, dass die LUT-Tiefe direkten Einfluss auf die Signalgüte hat. Ein Klirrfaktor unter 5 % wird mit mindestens 16 Stützstellen in der LUT erreicht. Nach Gleichung 4.2 werden bei 1 kHz und einem aktiven Kanal $\frac{15\text{ MHz}}{2 \cdot 1\text{ kHz}} = 7500$ Stützstellen pro Periode ver-

wendet. Es werden alle Stützstellen aus der LUT (maximal $\frac{7500}{4} = 1875$ Stützstellen) verwendet. Deshalb hat die Anzahl der LUT-Einträge viel Einfluss auf den Klirrfaktor.

Bei dem 150 kHz Signal wird dieser Effekt nicht so deutlich. Es reichen bereits 4 Stützstellen in der LUT, um einen Klirrfaktor unter 5 % zu erhalten. Eine größere LUT stellt keine signifikante Verbesserung der Signalgüte dar und bleibt konstant bei ca. 3,9 %. Die Anzahl der verwendeten Stützstellen, die pro Periode an den DAC geschickt werden, beträgt $\frac{15\ \text{MHz}}{2 \cdot 150\ \text{kHz}} = 50$. Somit wird ab einer LUT-Tiefe mit $\frac{50}{4} \approx 13$ Einträgen keine signifikante Verbesserung erreicht. Das Diagramm zeigt aber bereits bei 4 und 8 Stützstellen keinen wesentlichen Unterschied. Dies lässt sich durch Verzerrungen des DACs erklären.

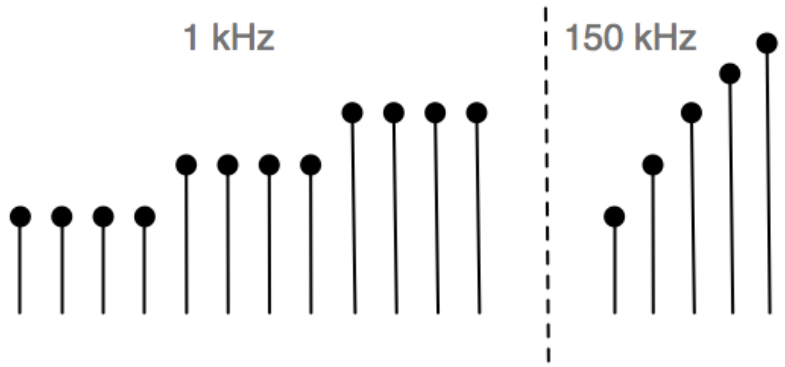


Abbildung 4.5.: Amplitudenausgabe bei 1 kHz und 150 kHz

Wie in Abbildung 4.5 schematisch veranschaulicht, werden für niederfrequente Ausgabesignale einzelne LUT-Einträge mehrfach nacheinander ausgegeben. Das Modul des DACs sendet nur neue Amplituden an den DAC, sodass der DAC damit effektiv mehr Zeit hat, den Spannungswert korrekt einzustellen. Bei höherfrequenten Signalen müssen hingegen größere Spannungssprünge in kürzerer Zeit dargestellt werden. Der DAC benötigt dafür aber eine gewisse Zeit (Einschwingzeit, 6 μs für einen Sprung von 7F00h auf 8100h [Texa09]). Während dieser Zeit werden bereits neue Werte an den DAC geschickt, die konvertiert werden sollen. Dadurch liegt zu keiner Zeit ein konstanter Wert an. Der DAC "*verzieht*" die Spannungen, wodurch sich keine Stufen im Ausgangssignal einstellen. Aus diesem Grund hat die LUT-Tiefe bei der Erzeugung des 150 kHz Signals keinen so großen Einfluss auf den Klirrfaktor, wie es bei dem 1 kHz Signal der Fall ist.

Für die maximale Genauigkeit (eine ausführlichere Analyse hierfür ist in Abschnitt 4.6 zu finden) wird die LUT-Breite der DAC-Auflösung von 16 bit angepasst. Der verwendete FPGA *M1AGL1000V2* besitzt Single-Port BRAMs mit 512x18 Bit Größe. Die im Diagramm 4.4 dargestellten Speichertiefen sind daher alle mit einem einzelnen BRAM realisierbar und haben somit keinen Einfluss auf die benötigten Logikressourcen.

4.5 Interpolation zwischen Stützstellen

Eine lineare Interpolation zwischen den Stützstellen ist gerade bei einer LUT mit wenigen Einträgen effektiv. Abbildung 4.6 zeigt die Auswirkung der Interpolation bei 1 kHz. Der FPGA wurde aufgrund der vom Synthese-Tool geschätzten Maximalfrequenz bei 7 MHz betrieben.

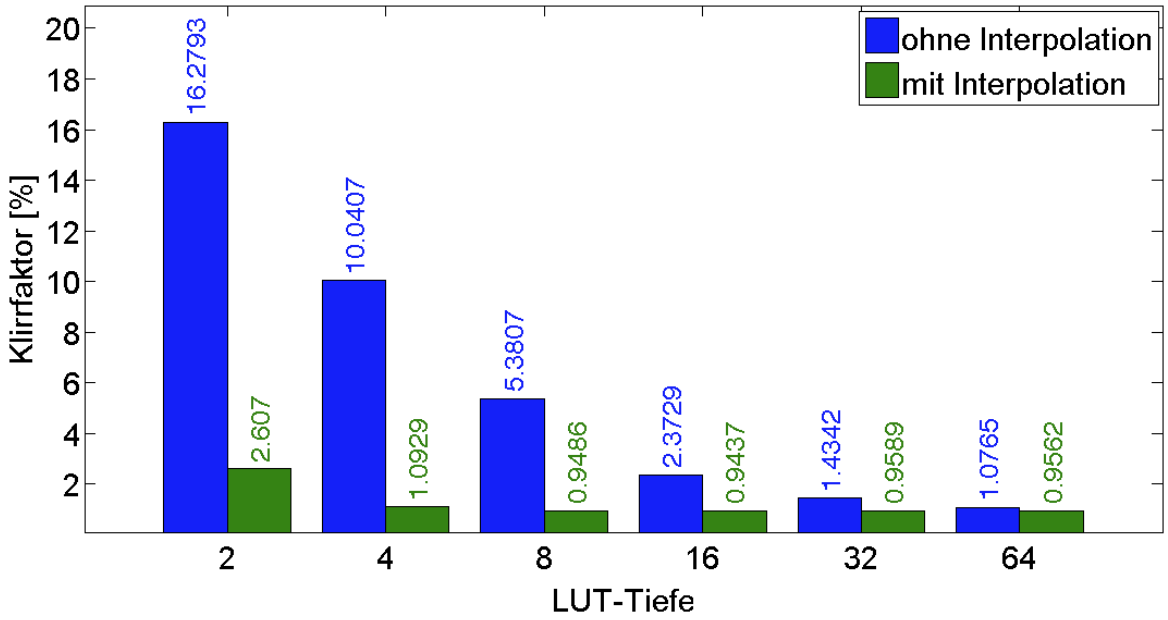


Abbildung 4.6.: Lineare Interpolation bei 1 kHz

Die Abbildung zeigt eine Verbesserung der Signalgüte unter Verwendung der Interpolation. Der Effekt der Interpolation wird bei einem 1 kHz Signal sehr deutlich. Die ursprünglich lineare Verschlechterung des Klirrfaktors bei kleiner werdender LUT-Tiefe wird durch die lineare Interpolation nahezu kompensiert. Bei einem aktiven Kanal mit einer Frequenz von 150 kHz verbessert sich der Klirrfaktor lediglich bei einer LUT-Tiefe von 2. Ohne Interpolation beläuft sich der Klirrfaktor auf ca. 4,5 %, mit Interpolation sind es nur ca. 3,4 %. Ansonsten weist er keine signifikanten Unterschiede auf. Dieser Effekt lässt sich auf die verzögerte Ausgabe des DACs zurückführen, wie bereits in Kapitel 4.4 erläutert.

Die Interpolation verbraucht durch die Multiplikation mit dem *Interpolationsfaktor* viele Ressourcen. Die benötigten Logikeinheiten bei einem aktiven Kanal belaufen sich auf ca. 4000. Dies entspricht einer Auslastung von 16,3 %. Eine mögliche Taktfrequenz liegt bei ca. 6 MHz. Dabei wurde der RAM mithilfe des CORDIC-Algorithmus initialisiert. Ohne eine Interpolation sind es nur etwa 7,1 % bei einer Frequenz von 18,4 MHz. Das FPGA besitzt Dual-Port BRAMs mit einem Speicherplatz von 4kx9 Bit. Bei einer LUT-Breite von 16 Bit benötigt die Interpolation zwei Dual-Port fähige BRAMs.

Im Vergleich dazu betrachten wir die Interpolation mittels zweier CORDIC Pipelines. Diese liefern die interpolierten Werte mit der Geschwindigkeit der Dual-Port BRAM Konfiguration, benötigen mit knapp 9000 CoreCells (36,6 %) aber auch mehr

als doppelt soviele Logikeinheiten. Die von der Synthesesoftware geschätzte Frequenz liegt bei 5,8 MHz. Diese Konfiguration benötigt jedoch keine BRAMs.

4.6 Auswirkung der LUT-Breite

Die Breite der LUT entscheidet über die Genauigkeit der Amplituden. Wenn die Breite der LUT von der Breite des DACs abweicht, müssen die Amplituden auf die Breite des DACs angepasst werden. Sie werden skaliert mit dem Faktor aus Gleichung 4.3.

$$scaleAmplitude = \frac{2^{WIDTH\_DAC}}{2^{WIDTH\_SAMPLES}} \tag{4.3}$$

Wenn die Bitbreite der Datensätze höher als die Bitbreite des DACs ist, verlieren die Werte durch die Skalierung an Genauigkeit. Wenn $WIDTH\_SAMPLES < WIDTH\_DAC$ gilt, sind die Werte in der LUT ungenauer und sollten sich auf die Signalgüte auswirken. Messungen haben gezeigt, dass der Klirrfaktor keine signifikanten Verschlechterungen bei einer geringeren LUT-Breite aufweist. Es wurden Bitbreiten von 16 bis 9 analysiert. Mit einer noch kleineren Bitbreite lassen sich auf der verwendeten Testkomponente keine Ressourcen einsparen. Der Single-Port BRAM ist maximal 18 Bit breit, der Dual-Port BRAM 9 Bit. Unter Verwendung der Interpolation mit einem RAM und der daraus resultierenden Notwendigkeit eines Dual-Port BRAMs lässt sich mit einer LUT-Breite von 9 Bit ein BRAM sparen. Selbst Messungen bei 1 kHz, welche Konfigurationsunterschiede sehr gut sichtbar machen, haben nur einen geringen Unterschied in der Signalgüte gezeigt. Der Klirrfaktor-Unterschied zwischen einer Bitbreite von 9 und 16 liegt bei ca. 0,1 %.

4.7 Auswirkungen der Anzahl der CORDIC-Iterationen

Die Genauigkeit der vom CORDIC-Algorithmus berechneten Amplitudenwerte steigt mit der Anzahl der ausgeführten Iterationen. Wieviele Iterationen tatsächlich notwendig sind, wird im Folgenden diskutiert.

4.7.1 Genauigkeit der berechneten Amplituden

Zunächst werden die per CORDIC-Algorithmus ermittelten Amplitudenwerte mit dem idealen Verlauf der auf 16 Bit Ganzzahlen normierten Sinusfunktion verglichen.

In Abbildung 4.7 ist der relative mittlere quadratische Fehler der CORDIC-Berechnungen bei verschiedenen Iterationstiefen dargestellt.

Dieser sinkt mit steigener Iterationsanzahl von 16,0383 % bei einer Iteration auf 0,01685 % bei 15 Iterationen. Bereits ab 6 Iterationen beträgt die Abweichung weniger als ein Prozent.

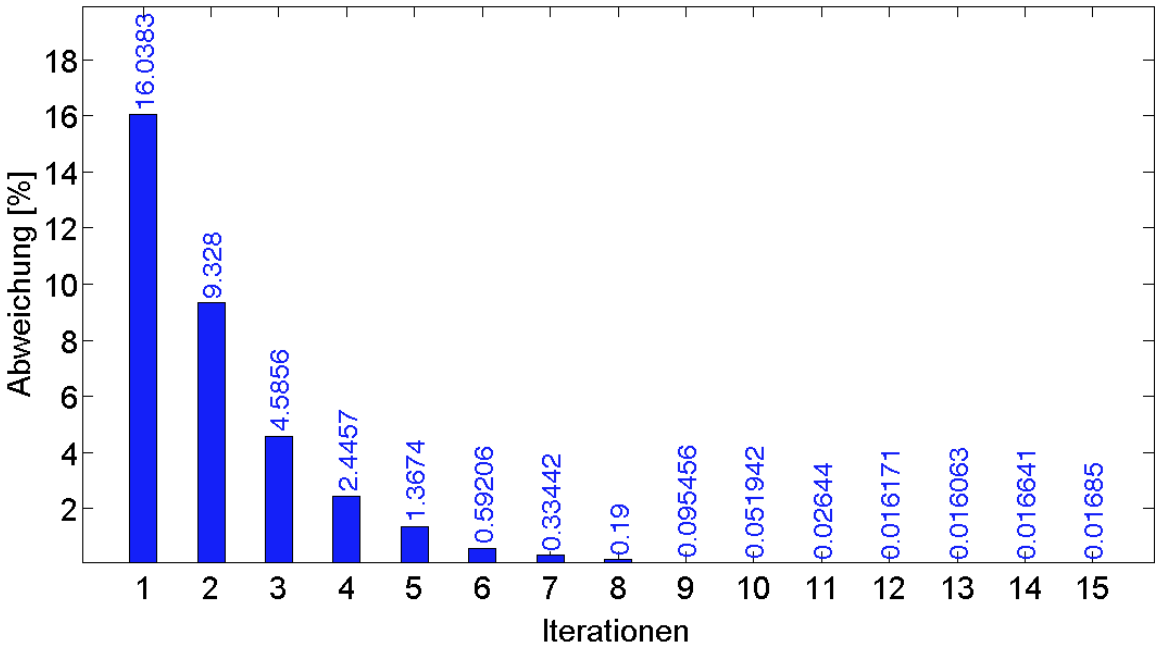


Abbildung 4.7.: Abweichung zu exakten Werten bei unterschiedlicher Iterationsanzahl

4.7.2 Auswirkungen der Iterationsanzahl auf die Signalgüte

Da man von der Genauigkeit der per CORDIC-Algorithus berechneten Amplitudenwerte nicht direkt auf die Qualität des erzeugten Signals schließen kann, wird diese hier noch einmal gesondert untersucht.

Abbildung 4.8 zeigt den Verlauf des Klirrfaktors für verschiedene Iterationstiefen während der LUT-Initialisierung. Hierbei wird der Verlauf des Klirrfaktors bei einem

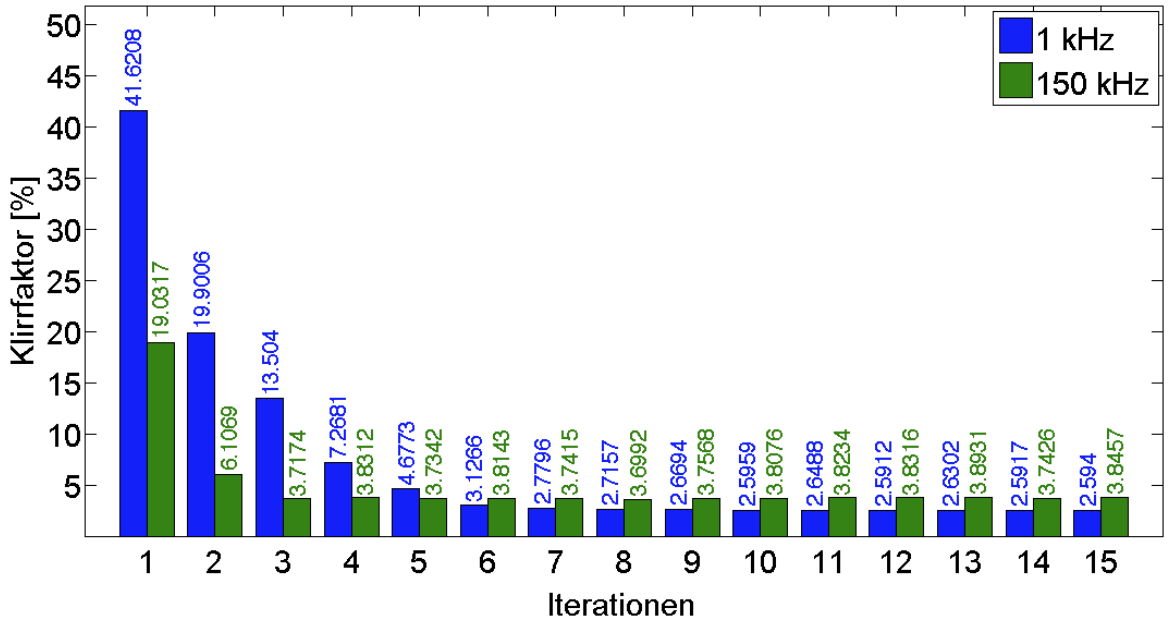


Abbildung 4.8.: Klirrfaktor in Bezug zu der Anzahl an Iterationen

aktiven Kanal mit 150 kHz und 64 Stützstellen in der LUT beschrieben. Der Klirrfaktorverlauf des 1 kHz Sinussignals wurde bei einer LUT-Tiefe von 16 erzeugt. In beiden Fällen wurde das FPGA mit 15 MHz getaktet.

Der Zusammenhang zwischen der Iterationstiefe und der resultierenden Signalqualität ist bei einem 1 kHz Signal stärker ausgeprägt als bei hohen Frequenzen, da die Ungenauigkeit der Amplitudenberechnung durch die DAC-Verzerrung (vgl. Abschnitt 4.4) maskiert wird. Dies wird durch den Unterschied in den zwei Verläufen sichtbar.

Wie bereits im vorhergehenden Kapitel gesehen, liefern zu wenige Iterationen zu ungenaue Werte. Dies spiegelt sich in Abbildung 4.8 durch einen höheren Klirrfaktor wieder. Bei einem 1 kHz Signal sind mindestens 5 Iterationen nötig, damit sich ein Klirrfaktor unter 5 % einstellt. Niedrigere Iterationsanzahlen lassen den Klirrfaktor annähernd exponentiell steigen. Bei einem 150 kHz Signal ist dieser Verlauf nur vergleichsweise schwach ausgeprägt.

Bei diesen Messungen wurde eine LUT verwendet. Der CORDIC-Algorithmus arbeitete ohne Pipeline, wodurch der Ressourcenverbrauch nur unwesentlich von der Iterationsanzahl abhängt.

4.7.3 Amplitudenberechnung zur Laufzeit

In diesem Kapitel wird untersucht, wie sich die Berechnung der Amplitude zur Laufzeit auf die Signalqualität und den Resourcenbedarf auswirkt. Hinsichtlich der Signalqualität unterscheidet sich die CORDIC-Pipeline nicht vom LUT-Verfahren, da beide Methoden einen Signalwert pro Takt ausgeben können. Deshalb wird an dieser Stelle nicht näher auf die Signalqualität der Pipeline-Konfiguration eingegangen.

Die Pipeline hat allerdings einen immensen Verbrauch an Logikeinheiten. Abbildung 4.9 verdeutlicht dies. Pro Iteration wird eine Iterationsstufe mit Addierern, Shiftern

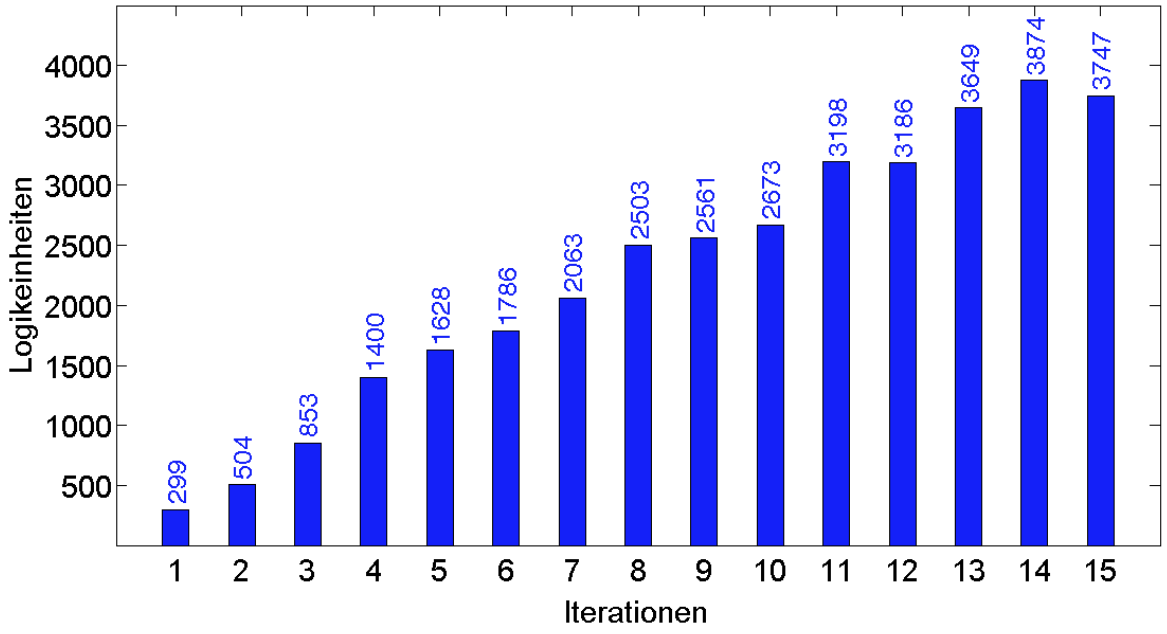


Abbildung 4.9.: Verbrauch an Logikeinheiten bei einer Pipeline Konfiguration

und Registern benötigt. Deshalb kommt es - zumindest annähernd - mit steigender Iterationsanzahl zu einem höheren Ressourcenverbrauch.

Die Amplituden lassen sich auch ohne Pipeline zur Laufzeit berechnen. Die Anzahl der CORDIC-Iterationen bestimmt dann die Taktzyklen, die zur Ausgabe eines Signalwertes notwendig sind. Mit zunehmender Iterationstiefe steigt dann zwar die Genauigkeit der berechneten Amplituden (vgl. Abschnitt 4.7.1), andererseits können weniger Stützstellen pro Signalperiode generiert werden. Gleichung 4.4 verdeutlicht den Zusammenhang zwischen den erzeugten Stützstellen pro Periode, die tatsächlich im Ausgangssignal enthalten sind, und der Iterationsanzahl.

$$samples = \begin{cases} \frac{FREQ\_DIGITAL}{STEPS\_DAC \cdot CHANNEL\_AMOUNT \cdot f_{out}} & \text{wenn } CORDIC\_ITERATIONS < STEPS\_DAC \\ \frac{FREQ\_DIGITAL}{CORDIC\_ITERATIONS \cdot CHANNEL\_AMOUNT \cdot f_{out}} & \text{wenn } CORDIC\_ITERATIONS \geq STEPS\_DAC \end{cases} \tag{4.4}$$

Bei 15 Iterationen, einem aktiven Kanal und einer Controller-Frequenz von 15 MHz ergeben sich nur noch $samples = \frac{15\ \text{MHz}}{15 \cdot 150\ \text{kHz}} = 6,\overline{6}$ Stützstellen, mit denen das Signal erzeugt wird. Dies sind deutlich weniger, als bei einer LUT-Methode ($\frac{15\ \text{MHz}}{2 \cdot 150\ \text{kHz}} = 50$).

In Abbildung 4.10 ist der Klirrfaktor in Bezug zur Iterationsanzahl des sequentiellen CORDIC-Algorithmus dargestellt. Es wurde nur ein aktiver Kanal betrieben mit 150 kHz und einer Phasenauflösung von 6 Bit. Die DDS wurde mit 15 MHz getaktet.

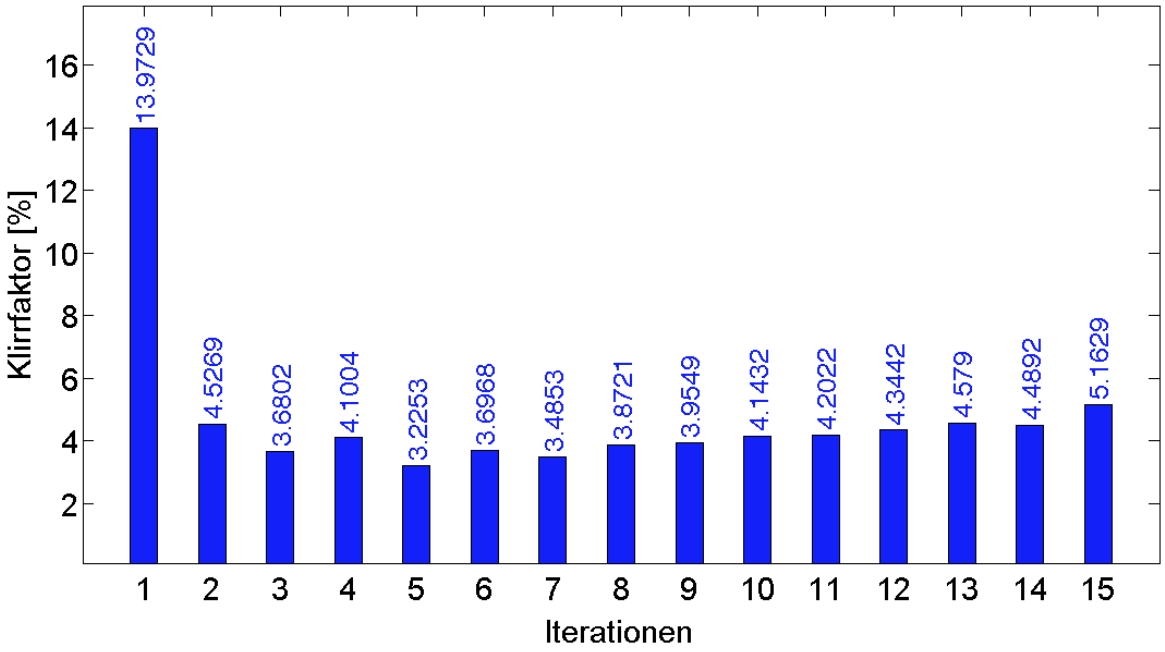


Abbildung 4.10.: Auswirkung der Iterationsanzahl bei einer Amplitudenberechnung zur Laufzeit

Bei 15 Iterationen liegt der Klirrfaktor bei etwa 5 %. Bei weniger Iterationen können mehr Stützstellen verwendet werden. Deshalb sinkt der Klirrfaktor bei geringerer Iterationsanzahl, obwohl die Genauigkeit der generierten Amplituden abnimmt.

Der Klirrfaktor wird im Bereich von fünf bis sieben Iterationen minimal. Der Klirrfaktor liegt dort sogar unter dem Klirrfaktor-Minimum einer LUT-Variante und lässt sich durch Messungenauigkeiten erklären.

Wenn mit der DDS mehrere Kanäle ohne RAM und ohne CORDIC-Pipeline verwendet werden, verschlechtert sich der Klirrfaktor. Bei zwei Kanälen mit 150 kHz, 15 Iterationen und einer Controller-Frequenz von 15 MHz kann die DDS nach Gleichung 4.4 nur $\frac{15\ \text{MHz}}{15 \cdot 2 \cdot 150\ \text{kHz}} = 3,\overline{3}$ Stützstellen pro Kanal und pro Periode ausgeben. Das resultiert in einem Klirrfaktor von 6,0872 %. Bei einem Kanal sind es nach Abbildung 4.10 5,1629 %. Wenn allerdings 3 Iterationen genutzt werden, lassen sich $16,\overline{6}$ Stützstellen pro Periode und pro Kanal generieren. Der Unterschied zwischen den Klirrfaktoren von einem Kanal auf zwei Kanäle liegt nur bei ca. $+0,1$ %.

Eine DDS Implementierung ohne RAM ist somit durchaus möglich und vor allem auch rentabel. Die Anzahl der benötigten Logikeinheiten bei einer Konfiguration ohne RAM ist ähnlich der mit RAM. Man spart jedoch die BRAM Blöcke.

Für kleinere Frequenzen ist dieser Ansatz sehr sinnvoll. Bei einem 1 kHz Signal lässt sich kein Unterschied in der Signalgüte im Vergleich zu einer Look-Up Methode finden.

4.8 Tiefpassfilter

In diesem Kapitel wird die Funktion eines Tiefpassfilters anhand eines Signals analysiert, welches mit wenigen Stützstellen erzeugt wird.

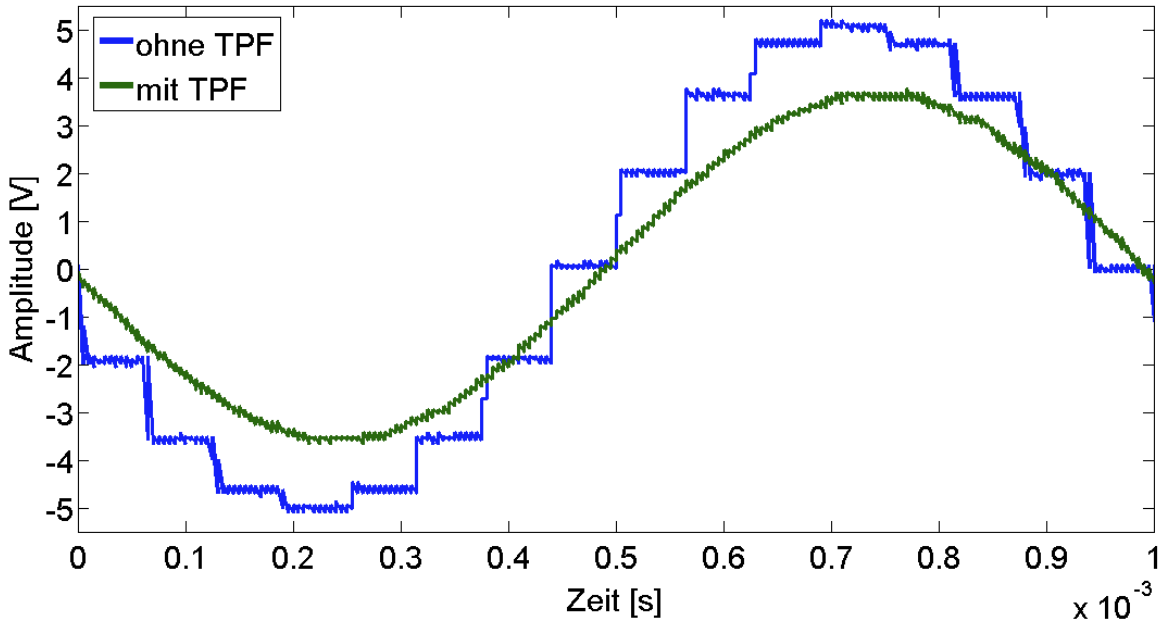


Abbildung 4.11.: Zeitsignal eines 1 kHz Signals mit anschließender Filterung

Abbildung 4.11 zeigt das Zeitsignal eines stark quantisiertem 1 kHz Signals (blau) und das Zeitsignal nach einer Tiefpassfilterung erster Ordnung (grün). Die Signale wurden mit 4 Stützstellen in der LUT erzeugt. Das in dem quantisiertem Signal überlagerte Rechtecksignal hat nach Gleichung 3.39 eine Frequenz von 16 kHz. Zur Eliminierung dieser ungewünschten Quantisierungseffekte wird das Signal mit einem

Tiefpassfilter erster Ordnung gefiltert. Bei einer Dimensionierung der Filterkomponenten zu $R = 7,2$ kΩ und $C = 22$ nF resultiert nach Gleichung 3.48 eine Grenzfrequenz von 1004,8 Hz. Der Tiefpassfilter wurde ohne aktive Verstärker verwendet, sodass die Amplitude des Ausgangssignals gedämpft wird. Der Tiefpassfilter dämpft das Rechtecksignal, sodass es nicht mehr erkennbar ist. Dadurch wird eine höhere Signalqualität erreicht, wie nachfolgend ebenfalls anhand des Klirrfaktors gezeigt wird.

Abbildung 4.12 zeigt den Klirrfaktor des gefilterten und des ungefilterten Signals in Bezug auf die Tiefe der LUT. Die Messungen beziehen sich auf ein 1 kHz Signal, welches bei einer FPGA Frequenz von 15 MHz erzeugt wurde.

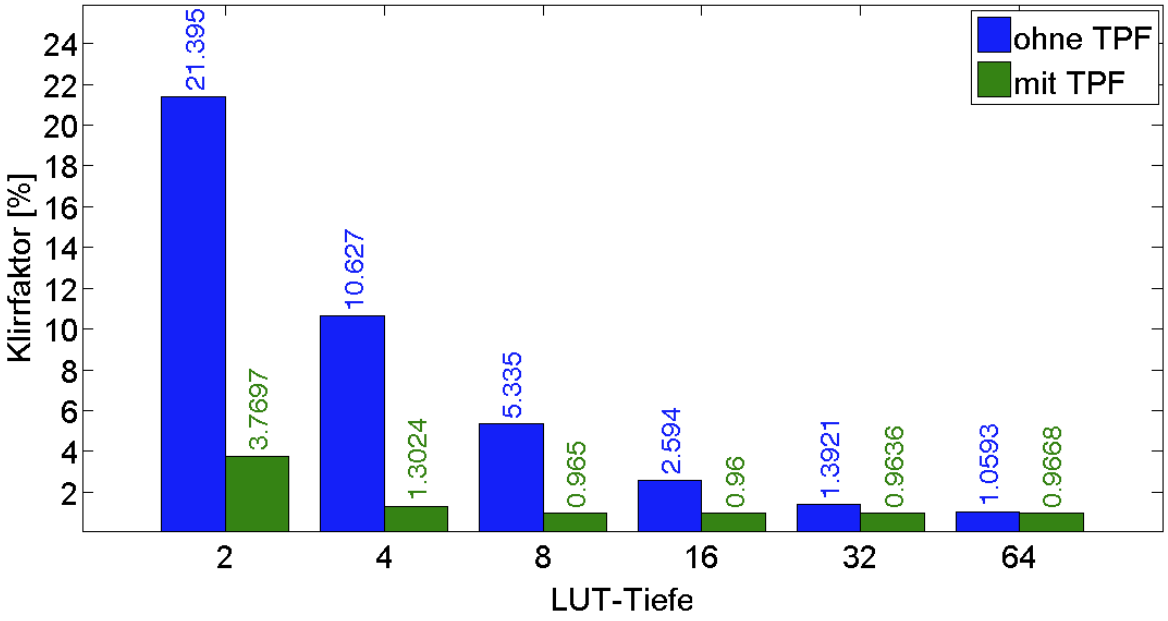


Abbildung 4.12.: Signalgüte eines 1 kHz Signals unter Verwendung eines TPF erster Ordnung

Durch einen TPF erster Ordnung kann mit vergleichsweiße geringem Aufwand eine große Verbesserung der Signalqualität erzielt werden. Bei wenigen Stützstellen in der LUT ist der Klirrfaktor hoch, wie bereits in Kapitel 4.4 gezeigt wurde. Bei einer LUT-Tiefe von zwei entsteht durch den TPF eine Klirrfaktor-Verbesserung von 21,4 % auf 3,8 %. Wenn mehr Stützstellen in der LUT vorhanden sind, nähern sich die Klirrfaktoren an. Bereits mit 8 Stützstellen in der LUT wird nach einer Tiefpassfilterung die Signalqualität des kommerziellen Signalgenerators erreicht (vgl. Abschnitt 4).

Die Analyse bezieht sich nur niederfrequente Signale bei denen die Quantisierungsstufen der LUT sichtbar sind. Bei höherfrequenten Signalen werden diese Quantisierungsstugen bereits durch die Trägheit des DAC verzerrt (vgl. Abschnitt 4.4). Dennoch lässt sich mit einem Tiefpassfilter die Signalqualität eines 150 kHz Signals verbessern, wie in Abbildung 4.13 ersichtlich. Der hier verwendete Tiefpassfilter wurde nach Gleichung 3.48 mit $R = 48\ \Omega$ und $C = 22$ nF für eine Grenzfrequenz $f_g \approx 150714$ Hz dimensioniert.

In diesem Fall resultiert die Verbesserung des Klirrfaktors in erster Linie aus der Dämpfung hochfrequenter Störanteile, welche durch Ungenauigkeiten in der Signal-

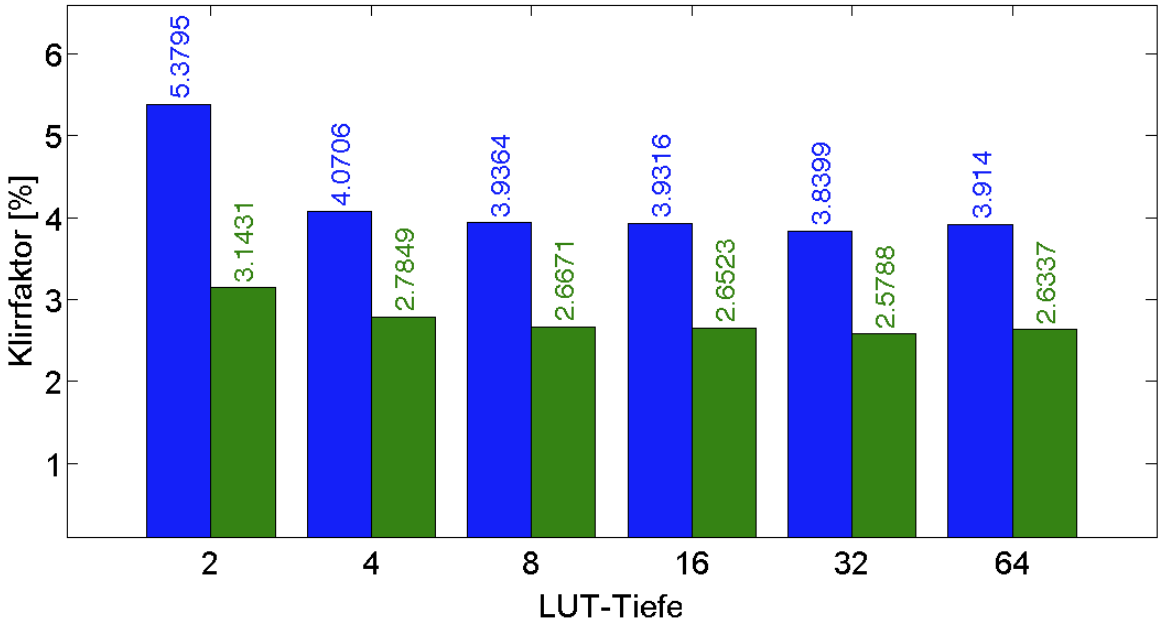


Abbildung 4.13.: Signalgüte eines 150 kHz Signals unter Verwendung eines TPF erster Ordnung

verarbeitungskette entstanden sind. Der DAC verzerrt ein Sinussignal gerade bei 150 kHz, wie bereits in 4.4 erläutert wurde. Durch diese ungenaue Darstellung des Sinussignals können Oberwellen entstehen, welche mit dem Tiefpassfilter gedämpft werden.

4.9 Genauigkeit der Ausgangsfrequenz

Die Qualität der Signale wurde bisher nur mit dem Klirrfaktor bewertet. Dieser verdeutlicht Ungenauigkeiten in der Signalform.

Im Folgenden wird die Genauigkeit der Ausgangsfrequenz mithilfe des Matlab-Skripts aus A.2.2 beurteilt. Dabei betrachten wir Frequenzen in einem Bereich von 1 kHz bis 2 kHz. Bei Signalen mit 150 kHz lassen sich mit der hier verwendeten Messmethodik keine eindeutigen Aussagen zur Genauigkeit zeigen.

Wie bereits erwähnt lässt sich mit dem Parameter *FREQ_OUT_RESOLUTION* die Frequenzauflösung einstellen, welche aufgrund von Rundungen geringer als angegeben sein kann.

Wir betrachten den Fall bei einer konfigurierten Frequenzauflösung von 10 Hz und einer Akkumulator-Frequenz von 15 MHz. Nach Gleichung 3.4 ergibt sich die Akkumulatorbitbreite zu $\overline{N} = 21$. Der Sprungwert *base* zu der unteren Frequenz des hier gewählten Frequenzbandes berechnet sich nach Gleichung 2.5 bei einer Ausgangsfrequenz von $f_{out} = 1$ kHz zu $base = \frac{1\text{ kHz} \cdot 2^{21}}{15\text{ MHz}} = 139$.

Nach Gleichung 3.5 ergibt sich mit $sum = 1$ eine Ausgangsfrequenz von 1001,36 Hz. Für einen um Eins erhöhten Sprungwert beträgt die Ausgangsfrequenz 1008,51 Hz. Die Schrittweite liegt damit bei 7,15 Hz und somit unter der angegebenen minimalen Frequenzauflösung.

Diese theoretischen Werte werden in Tabelle 4.3 mit Frequenzen aus Messungen verglichen. Die gemessenen Frequenzen sind Durchschnittswerte aus allen in der Messung vorhandenen Perioden.

	Theoretische Frequenz	**Gemessene Frequenz**
$\boldsymbol{sum} = 1$	1001,36 Hz	1001,1 Hz
$\boldsymbol{sum} = 2$	1008,51 Hz	1008,2 Hz

Tabelle 4.3.: Genauigkeit der Ausgangsfrequenzen anhand theoretischen und gemessenen Werten

Die gemessenen Signale bestätigen die theoretischen Frequenzen und die Schrittweite der Eingangsvariablen *sum*.

5 Zusammenfassung und Ausblick

Die vorliegende DDS Implementierung wurde so gekapselt, dass Erweiterungen leicht eingebunden werden können. Diese Version erlaubt eine LUT-Initialisierung mittels externem Flash-Speicher oder dem CORDIC-Algorithmus. Neben der LUT-Methode kann der CORDIC-Algorithmus verwendet werden, um die Amplituden zur Laufzeit zu berechnen. Dabei wurde eine gezeitmultiplexte und eine Pipeline-Variante implementiert. Durch die Pipeline wird zu jedem Systemtakt eine Amplitude ausgegeben, wie es bei einer LUT-Methode der Fall ist.

Die Implementierung wurde weiterhin durch eine Approximation des aktuellen Amplitudenwertes erweitert. Die lineare Interpolation ermöglicht gute Resultate selbst bei wenigen LUT-Einträgen.

Durch die vielen Konfigurationsmöglichkeiten lässt sich diese DDS Implementierung an viele anwendungsbedingte Spezifikationen individuell anpassen. Durch diese gezielten Einstellungsmöglichkeiten kann eine ressourcenschonende und performante Signalsynthese erzielt werden.

Mit den verwendeten Komponenten können die Anforderungen aus Kapitel 1.2 umgesetzt werden. Die geforderte Anzahl an Kanälen kann mit dem Controller realisiert werden. Der Frequenzbereich um 150 kHz wird erreicht.

Verbesserungen des Klirrfaktors wären vor allem mit einem schnelleren DAC möglich. Die Einschwingzeit begrenzt die Qualität der per DDS erzeugten Signale bei Frequenzen um 150 kHz enorm. So erbringt die lineare Interpolation bei einem 150 kHz keine signifikanten Vorteile im Vergleich zu den verbrauchten Ressourcen. Eine LUT mit mehr Einträgen hat dadurch ebenfalls keine signifikanten Auswirkungen auf die Signalgüte.

Nach dem Nyquist-Kriterium wird mindestens die doppelte Frequenz des im Nutzsignal höchst vorkommenden Frequenzanteils benötigt, um das Signal nach einer Tiefpassfilterung zum ursprünglichen Signal zu rekonstruieren. [Brau04] Bei 15 MHz und dem hier gewählten DAC ergibt sich eine maximal mögliche Frequenz von 3,75 MHz. Dabei wird jedoch nur die Verwendung eines aktiven Kanals betrachtet. Eine doppelte Ausgabefrequenz nach Nyquist bedeutet, es werden nur zwei Amplituden pro Periode an den DAC geschickt. In der Praxis wird die Ausgabefrequenz höher gewählt, weil Tiefpassfilter mit hoher Ordnung notwendig sind, welche komplex und teuer sind. Richtwerte sind ca. 7 bis 10 Stützstellen pro Periode. [Matt11] Für die betrachteten Zielvorgaben von drei Kanälen und einer sehr kleinen LUT-Tiefe von zwei (das sind $2 \cdot 4 = 8$ Stützstellen, die pro Periode ausgegeben werden) beläuft sich die Maximalfrequenz auf 312,5 kHz pro Kanal.

Zur Vereinfachung wurden äquidistante Stützstellen verwendet. Eine Verbesserung wäre mithilfe von nicht-äquidistanten Stützstellen möglich. Bei den Amplitudenmaxima einer Sinusfunktion könnten weniger Stützstellen verwendt werden als bei dem Verlauf um Null. So könnte dieselbe Anzahl an Stützstellen effektiver genutzt werden, um das Sinussignal darzustellen.

Weil der DAC von der DDS separiert ist, kann das Modul des DACs mit einer anderen Taktfrequenz arbeiten als die DDS. Der hier verwendete DAC benötigt zwei Taktzyklen, um einen Wert zu schreiben. Wird der DAC mit der doppelten DDS-Frequenz betrieben, lässt sich die Anzahl der benötigten DAC-Zyklen über den Parameter *STEPS_DAC* auf Eins reduzieren. Damit lässt sich die doppelte Frequenz beim Ausgeben der Stützstellen erreichen. Auf dem hier verwendeten *M1AGL1000V2*-FPGA ist dies allerdings nur bedingt möglich, da die maximal mögliche Taktfrequenz für das DAC-Modul zu gering ist. Eine einfache Testanwendung zur Ausgabe von fortlaufenden Werten ermöglichte eine Taktfrequenz des DAC-Moduls von 20,6 MHz. Die DDS selbst kann abhängig von der Konfiguration ebenfalls bei 20 MHz getaktet werden, sodass eine Verdopplung des Takts im DAC-Modul nicht möglich ist. Allerdings verringert sich die Taktfrequenz der DDS bei bestimmten Einstellungen. So könnte in diesen Fällen eine Verdopplung des DAC-Takts verwendet werden. Außerdem könnte der DAC-Takt immer ca. 20 MHz betragen, dann könnte die DDS bei vielen Konfigurationen mit 10 MHz betrieben werden. Dies würde eine höhere Ausgaberate ermöglichen als er z.B. unter Verwendung einer CORDIC-Initialisierung möglich ist.

Es wurde angesprochen, dass der Flash-Speicher bei einer anderen Anzahl an Stützstellen in der LUT erneut beschrieben werden muss, was in einer Erprobungsphase aufwendig ist. Dies kann umgangen werden, indem eine hohe Anzahl an Stützstellen in dem Flash-Speicher abgelegt werden und bei einer Initialisierung nur die nötigen Einträge ausgelesen werden.

Literaturverzeichnis

[Acte09] Actel, Microsemi SoC Products Group. *IGLOOe Low Power Flash FPGAs with Flash*Freeze Technology, http://www.actel.com/documents/IGLOOe_DS.pdf, Revision 8, 2009*.

[Acte11] Actel, Microsemi SoC Products Group. *ARM Cortex-M1-Enabled IGLOO Development Kit User's Guide, http://www.actel.com/documents/M1IGLOO_DevKit_UG.pdf, Stand: 14.09.2011*.

[Agil11] Agilent Technologies. *Agilent 30 MHz Function/Arbitrary Waveform Generators (33522A), http://www.home.agilent.com/agilent/redirector.jspx?action=ref&cname=AGILENT_EDITORIAL&ckey=1919803&lc=ger&cc=DE&nfr=-536902257.940639.00, Stand: 10.10.2011*.

[Anal99] Analog Devices, Inc. *A Technical Tutorial on Direct Digital Synthesis, 1999*.

[Aure01] Aurelienr.com. *The Piezoelectric Effect, www.aurelienr.com/electronique/piezo/piezo.pdf, 2001*.

[Brau04] Martin Braun. *Studienarbeit: Entwurf und Implementierung eines FPGA basierten Synthesizerboards für Direkte Digitale Synthese, 2004*.

[EnLK11] Andreas Engel, Björn Liebig, Andreas Koch. *Feasibility Analysis of Reconfigurable Computing in Low-Power Wireless Sensor Applications, http://www.esa.informatik.tu-darmstadt.de/twiki/pub/Staff/AndreasKochPublications/2011_ARC_AEBL.pdf, 2011*.

[GaDS11] Prof. Kris Gaj, Gaurav Doshi, Hiren Shah. *Sine/Cosine using CORDIC Algorithm, http://teal.gmu.edu/courses/ECE645/projects_S06/talks/CORDIC.pdf, Stand: 05.09.2011*.

[Hahn91] Dipl.-Ing. Helmut Hahn. *Untersuchung und Integration von Berechnungsverfahren elementarer Funktionen auf CORDIC-Basis mit Anwendungen in der adaptiven Signalverarbeitung. VDI Verlag, Düsseldorf, Reihe 9 Nr. 125, 1991*.

[Hein11] Robert Heinemann. *PSpice-Simulation eines RC-Phasenschieber-Oszillators , http://www.spicelab.de/phasenschieber.htm, Stand: 09.10.2011*.

[Herb02] Heiner Herberg. *Elektronik - Einführung für alle Studiengänge, S. 21, Vieweg Verlag, Braunschweig/Wiesbaden, 1. Auflage, 2002*.

[KrSk08] Tobias Krähling, Tomislav Skoda. *Lock-In-Verstärker, http://www.semibyte.de/dokuwiki/nat/physik/studium_protokolle/102-lock-in-verstaerker, 2008*.

[Lehw11] Mario Lehwald. *Phasenschieber-Oszillator, http://www.hobby-bastelecke.de/grundschaltungen/oszillatoren_phasenschieber.htm, Stand: 09.10.2011*.

[Matt11] Prof. Dr. Wolfgang Matthes. *Theorie und Praxis der Signalabtastung, www.controllersandpcs.de/lehrarchiv/pdfs/elektronik/ees05_03.pdf, Stand: 18.10.2011*.

[Meli07] Dennis Melikjan. *Oszillatoren, http://www.fh-friedberg.de/fachbereiche/e2/telekom-labor/geissler/hf/arbeitsmaterial/pdf/HF_Kapitel_4.pdf, 2007*.

[Miet11] Detlef Mietke. *RC-Oszillator, http://www.elektroniktutor.de/signale/rc_osz.html, Stand: 07.10.2011*.

[Numo07] Numonyx. *Numonyx Embedded Flash Memory (J3 v. D), 2007*.

[Ohmb11] Ralf Richard Ohmberger. *Parallelschwingkreis, http://www.amplifier.cd/Tutorial/Grundlagen/Parallelschwingkreis.htm, Stand: 30.08.2011*.

[Riss04] Thomas Risse. *CORDIC-Algorithmen Verbinden Mathematik, Computer-Architektur und Anwendungen, www.wiete.com.au/journals/GJEE/Publish/vol8no3/Risse.pdf, 2004*.

[Sche08] Prof. Dr.-Ing. Gregor Schenke. *Skript zur Vorlesung "Industrieelektronik (Teil B)" an der Hochschule Emden/Leer, Fachbereich Technik, Abteilung Elektrotechnik & Informatik, http://www.et-inf.fho-emden.de/~elmalab/indelek/download/Ind_5.pdf, 2008*.

[Schl05] Prof. Dipl. El.-Ing. ETH Martin Schlup *Fourier-Reihe und -Spektrum, https://home.zhaw.ch/~spma/Scripts/SiSy_GSA/SiSy1/SiSy1_3_Fourierreihen/SiSy1_Th3_FourierReihen.pdf, 09.2005*.

[Schm11a] Prof. Dr.-Ing. Heinz Schmidt-Walter. *Vorlesungsskript: Elektronik, Kapitel 16 Aktive Filter, http://schmidt-walter.eit.h-da.de/el/skript_pdf/el_16.pdf, Stand: 17.08.2011*.

[Schm11b] Prof. Dr.-Ing. Heinz Schmidt-Walter. *Vorlesungsskript: Elektronik, Kapitel 15 Oszillatoren, http://schmidt-walter.eit.h-da.de/el/skript_pdf/el_15.pdf, Stand: 07.10.2011*.

[Stan11] Stanford Research Systems. *Direct Digital Synthesis - Application Note #5, Stand: 19.05.2011*.

[Stef07] Prof. Dr. B. Steffenhagen. *Formelsammlung Regelungstechnik, Fachhochschule Stralsund, Fachbereich Elektrotechnik und Informatik, http://www.fh-stralsund.de/dokumentenverwaltung/dokumanagement//35/copy46ef79649f541.pdf, 2007*.

[Syno01] Synotech Sensor und Meßtechnik GmbH. *Charakteristiken von Tiefpassfiltern, www.synotech.de/images/upload/KEM_AN01_061016.pdf, 2001*.

[Texa09] Texas Instruments. *DAC8728 Datenblatt, 2009*.

[Texa10] Texas Instruments. *DAC8728EVM User Guide, 2010*.

[Umni11] Umnicom.de. *Filter, http://www.umnicom.de/Elektronik/Schaltungssammlung/Filter/Ordnung1/Filter1.html, Stand: 14.09.2011*.

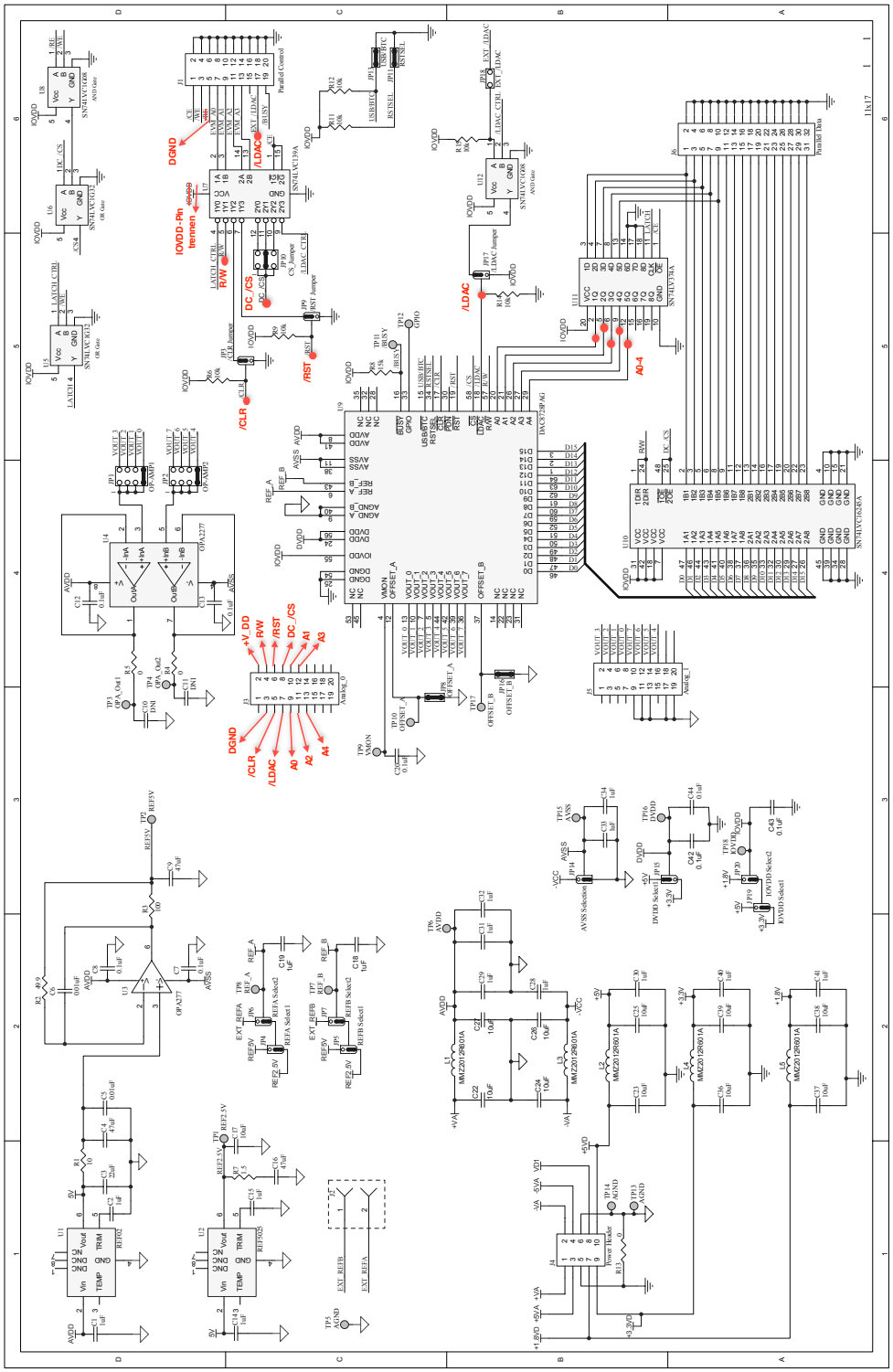
[Weiß05] Wilfried Weißgerber. *Elektrotechnik für Ingenieure 3, Vieweg Verlag, Wiesbaden, 5. Auflage, 2005*.

[Xili05] Xilinx. *DDS v5.0, http://www.xilinx.com/support/documentation/ip_documentation/dds.pdf, 2005*.

Abbildungsverzeichnis

A Anhang

A.1 Modifiziertes DAC Entwicklungsboard



A.2 Matlab Skripte

A.2.1 Klirrfaktorbestimmung eines Sinussignals aus Messdaten

```
% file: klirrfaktor.m
% computes distortion factor

% input:
choosenFrequency = 0; % 1 for 1khz, 0 for 150khz
plotPowerSpectrum = 0;

% fft analysis
% http://faculty.olin.edu/bstorey/Notes/Fourier.pdf page 11
x = data(:,2); % extract CSV data
if choosenFrequency == 1
    fs = 400e3; % samplingrate at 1 kHz
else
    fs = 1/(data(2,1)-data(1,1)); % sampling frequency
end

N = length(x)/2; % number of data points in positive ...
    frequency half
p = abs(fft(x))/N; % absolute value of the fft
p = p(1:N).^2; % power

dfs = fs/2/N; % frequency resolution
freqs = [0:N-1]*dfs; % find the corresponding frequency in Hz

% peak detection
if choosenFrequency == 0
    [val,index] = max(p);
    base = index-1;
else
    [val,index] = max(p(2:length(p)));
    base = index; % number of samples between harmonics
end
fbase = dfs*base; % base freqency

if plotPowerSpectrum == 1
    % plot power spectrum
    semilogy(freqs, p);
    ylabel('power');
    xlabel('frequency (Hz)');
    axis([0 3.5*fbase 0 100]); % zoom in
end

% start distortion factor
N = floor(N/base);
if choosenFrequency == 1
    if N*base == length(p)
        N = N-1;
```

```
47      end
48  end
49
50  % compute distortion factor with peak detection
51  dbase = base/10;
52  freqs = fbase;
53  vals = val;
54  for i = 2:N
55      [val,index] = ...
            max(p(round(i*base-dbase):round(i*base+dbase)));
56      freqs = [freqs (index+i*base-base/4)*dfs];
57      vals = [vals  val];
58  end
59  klirrdata = [freqs' vals'];
60  klirr = sqrt(sum(vals(2:N))/sum(vals)); % definition of ...
        distortion factor
61  klirrInPercentPeak = klirr*100
62
63  % compute distortion factor static
64  freqs = (1:N)*fbase;
65  vals = p(1+round((1:N)*base));
66  klirrdata = [freqs' vals];
67  klirr = sqrt(sum(vals(2:N))/sum(vals)); % definition of ...
        distortion factor
68  klirrInPercentStatic = klirr*100
```

A.2.2 Frequenzbestimmung eines Sinussignals aus Messdaten

```
1   % file: freq.m
2   % computes the frequency
3
4   % input
5   f_looking_for = 1000; % 1029 for Q=100, sum=1; 1087 for ...
        Q=100, sum=2; 1001 for Q=10, sum=1; 1008 for Q=10, sum=2
6
7   mydt = 1/(400000); % at 1 khz
8   ii = round(1/mydt);
9   ii = round(ii/f_looking_for); % theoretical steps between one ...
        period
10  [aaaaa, start_index] = max(data(1:ii,2));
11  delt_i = ii/4; % step-range to look left and right
12  delt_i = round(delt_i - 0.1*delt_i); % -10%, so the range is ...
        always in the actual period
13  f_result = f_looking_for;
14  secondMin = 0;
15  secondMinIndex = 1;
16
17  % look for all periods in the signal
18  for j = 1:round(length(data(:,1))/ii)-2 % -2, so ...
        index+start_index is always valid
19      firstMin = secondMin;
```

```
20      firstMinIndex = secondMinIndex;
21      my_abs_x = abs(data(start_index + round(sum(ii)-delt_i): ...
            start_index+round(sum(ii) + delt_i),2));
22      [aaaaaa, iii] = max(max(my_abs_x) - my_abs_x); % get the ...
            max amplitude of the sine
23      my_min = data(round(sum(ii) - delt_i + iii-1),2);
24      secondMinIndex = round(sum(ii) - delt_i + iii-1);
25      diff = secondMinIndex-firstMinIndex; % actual steps ...
            between one period
26      if (j > 1)
27          [f_result] = [f_result 1/(diff*mydt)]; % add ...
                frequency to results
28      end
29      [ii] = [ii diff]; % add steps to result
30  end
31
32  f_average = sum(f_result(2:length(f_result)-1))/ ...
        length(f_result(2:length(f_result)-1)) % average frequency
```