OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

ELEKTRO- UND
INFORMATIONSTECHNIK

PROJEKTARBEIT

Hannes Bachl

FPGA basierter BPSK/QPSK Empfänger

22.02.2024

Fakultät: Elektro- und Informationstechnik
Studiengang: Bachelor Elektro- und Informationstechnik
Betreuung: Prof. Dr.-Ing. Florian Aschauer

Erklärung

1. Mir ist bekannt, dass dieses Exemplar der Projektarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
2. Ich erkläre hiermit, dass ich diese Projektarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum und Unterschrift

Vorgelegt durch:	Hannes Bachl
Matrikelnummer:	3321274
Studiengang:	Bachelor Elektro- und Informationstechnik
Betreuung:	Prof. Dr.-Ing. Florian Aschauer

Vorwort

Inhalt dieser Projektarbeit ist die Erstellung eines einfachen Software defined radio zum Empfang eines QPSK oder BPSK modulierten Signales. Als Basis kommt hier ein Field-programmable gate array mit angeschlossenem externen Analog-to-digital converter zum Einsatz.

Hauptaugenmerk soll dabei auf der eigenen Umsetzung der für ein SDR notwendigen Signalverarbeitungsalgorithmen sowie den Kommunikationsstrukturen, welche benötigt werden um einer Central Processing Unit die Steuerung der umgesetzten Schaltung zu ermöglichen, liegen.

Das Projekt soll als grundlegende Basis für, ein eventuell später noch umzusetzendes, komplexeres SDR dienen und ist deswegen, wo möglich, flexibel gestaltet.

Inhaltsverzeichnis

1 Einleitung

Inhalt dieses Projektes ist die Erstellung eines SDR, wobei ein FPGA als Plattform dienen soll. Der FPGA-Logikteil soll weiterhin von einem Rechenkern, für die weitere Datenverarbeitung, unterstützt werden.

Da es sich um ein Lernprojekt handele, sollte der Fokus, bei der Erstellung der FPGA-Schaltung, auf der selbständigen Erstellung aller Signal verarbeitenden Blöcke liegen. Komplexe Blöcke welche für die Kommunikation mit dem Rechensystem benötigt wurden *(z.B. AXI-Infrastruktur)* wurden aus Zeitgründen nicht selbst erstellt, sondern vorhandene IP-Kerne des FPGA-Herstellers verwendet.

1.1 Anforderungen

Zu beginn des Projektes wurde mit der Festlegung der Anforderungen an des Gesamtsystem begonnen. Folgend konnten die notwendigen Hardware- und Logikkomponenten richtig ausgelegt werden.

Dabei wurden die folgende Anforderungen an das System definiert:

1. **Unterstützung der Verwendung eines extern ADC**
   - 1.1. Abtastfrequenz von mindestens $100\,\mathrm{MHz}$ [$f_{ADC} \geq 100\,\mathrm{MHz}$].
   - 1.2. Datenbreite soll mindestens $12\,\mathrm{bit}$ betragen [$w_{adc} \geq 12$]
   - 1.3. Externer Antialias Filter notwendig, Abtastung in der ersten Nyquist-Zone.
2. **Digitale Abwärtsmischung in das Basisband**
   - 2.1. Eingangssignal im Frequenzbereich $1\,\mathrm{MHz}$ bis $20\,\mathrm{MHz}$. [$f_s \in [1\,\mathrm{MHz}; 20\,\mathrm{MHz}]$]
   - 2.2. Interner Numerically controlled oscillator, wobei die Frequenz innerhalb des Frequenzbereichs konfigurierbar sein soll
   - 2.3. Interner, komplexer Mischer mit $16\,\mathrm{bit}$-Breitem I/Q Signal am Ausgang
   - 2.4. Dezimierungsfilter mit, zum Synthesezeitpunkt einstellbaren, variablen Dezimierungsverhältnis
   - 2.5. Eine interne FPGA-Taktrate von $f_{clk} \geq 200\,\mathrm{MHz}$ soll unterstützt werden.
3. **Carrier-Tracking für BPSK und QPSK Signale**
   - 3.1. Interner Phasen-Komparator mit folgendem Regler zur Konditionierung des lokalen Oszillators.
   - 3.2. Modulation (BPSK/QPSK) soll zur Laufzeit wechselbar sein.

3.3. Als Regler soll ein PID-Regler mit, zur Laufzeit einstellbaren Koeffizienten, sein.

3.4. Das Carrier-Tracking soll Abschaltbar sein und nur aktiv sein wenn das Eingangssignal eine gewisse Amplitude aufweist.

4. **Kommunikation mit dem Rechenkern**

4.1. Ein Rechenkern soll die FPGA-Schaltung verwalten und die weitere Verarbeitung der Nutzdaten übernehmen.

4.2. Die Steuerung der einzelnen Module soll der Rechenkern über eine AXI-Lite Registerbank vornehmen können.

4.3. Die gemischten und dezimierten Nutzdaten sollen via Direct Memory Access-Controller dem Rechenkern zur Verfügung gestellt werden.

1.2 Auswahl der Hardware-Komponenten

Nach Festlegung der Anforderungen wurde die für die Umsetzung notwendige Hardware ausgewählt.

Aufgrund der Anforderung 4.1 sowie dem Fokus der Vorlesung, FPGA-Logik in CPU-Rechensysteme zu integrieren, ist es notwendig einen FPGA mit eingeschlossenen CPU-Kernen auszuwählen.

Die Wahl fiel hier aus Kosten- und da Know-How-Gründen sowie aufgrund der bereits Vorhandenen Tool-Umgebung, auf die Zynq-7000 APSoC Serie des Herstellers Xilinx. Es handelt sich hierbei um eine FPGA-Fabric der Xilinx 7-Series *(PL,Programmable-Logic)* mit angeschlossenem Dual-Core ARM Cortex-A9 *(PS,Processing-System)* als Rechenkern.

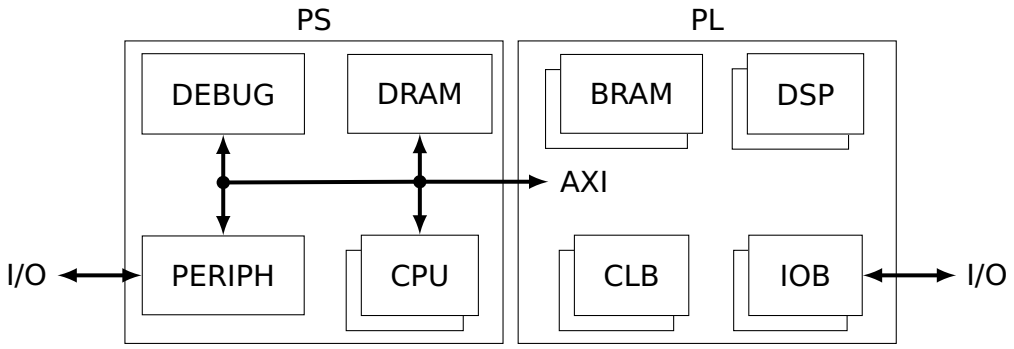

Abbildung 1.1: Struktur des verwendeten Zynq-7000 FPGA

Die in der FPGA-Fabric (PL-Teil) vorhandenen Logic-Resourcen *(BRAM,CLB,IOB, DSP-Slices)* werden gemäß des während der Implementierung erstellen FPGA-Designs verdrahtet und am Ende durch einen AXI-Bus mit dem Rechensysteme (PS-Teil) verbunden.

Da der FPGA mit ADC Modul selbst beschafft und bezahlt wird war hier vor allem der Preis ausschlaggebend. Die Wahl fiel hier auf ein Eclypse Z7 Board der Firma Digilent.

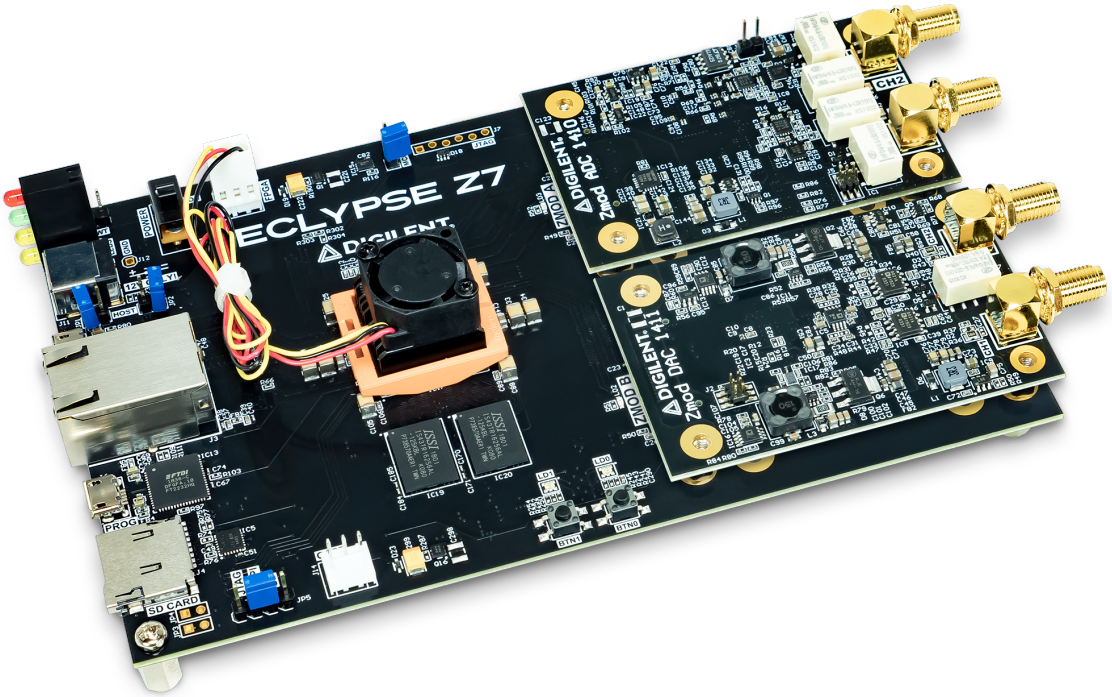

Abbildung 1.2: Bild des Eclypse-Z7 mit ADC und DAC
©Digilent Inc. [5]

Hauptgrund für die Auswahl dieses Boards war der, im Vergleich zu anderen FPGA Mezzanine Card (FMC) basierten Boards, relativ günstige Preis, sowie das Vorhandensein von vielen IP-Cores der Firma Digilent welche die Ansteuerung der einzelnen Hardware-Komponenten vereinfachen. Zusätzlich hilfreich war der große Umfang von verfügbaren Referenzmaterialien.

Der von dem Board verwendete ADC *(ZModScope 1410-105, AD9648BCPZ-105)* erfüllt alle gestellten Anforderungen.[6]

Bei dem verwendeten FPGA handelt es sich um einen XC7Z020-1CLG484C mit Dual-Core Cortex-A9 Prozessor mit $666\,\mathrm{MHz}$, sowie $1\,\mathrm{GiB}$ externem DDR3-DRAM[5]

Ein DAC wäre ebenfalls vorhanden, wird jedoch nicht genutzt.

2 Entwurf und Design

2.1 FPGA-Design

Nach Festlegung der Anforderungen wurde mit der Konzeption und Entwurfsphase des Gesamtsystemes sowie des FPGA-Designs begonnen.

Hierbei wurde zuerst, anhand der Anforderungen, die Architektur des zu erstellenden FPGA-Designs festgelegt.

Die Kommunikation zwischen den einzelnen Modulen, besonders für den Datenaustausch, soll zum größten Teil mit einer AXI-Stream basierten Schnittstelle umgesetzt werden. Für die, hauptsächlich zur Steuerung verwendete, Kommunikation zwischen FPGA-Design und Rechenkern sollen AXI-Lite basierte Registerbänke verwendet werden.

In dem folgenden Entwurf nicht betrachtet werden die weiterhin notwendigen AXI-Infrastruktur Blöcke, welche für die Kommunikation mit dem Rechenkern weiterhin benötigt werden. Diese Blöcke werden jedoch aus Zeit- und Komplexitätsgründen nicht selbst umgesetzt. Eine hierarchische Gesamtübersicht des Designs ,mit AXI-Infrastruktur, kann im Kapitel 3.1 gefunden werden.

Als Architektur für den Funkempfangsteil des Designs wurde eine klassische QPSK-Empfängerarchitektur mit direkter digitaler Abwärtsmischung gewählt.

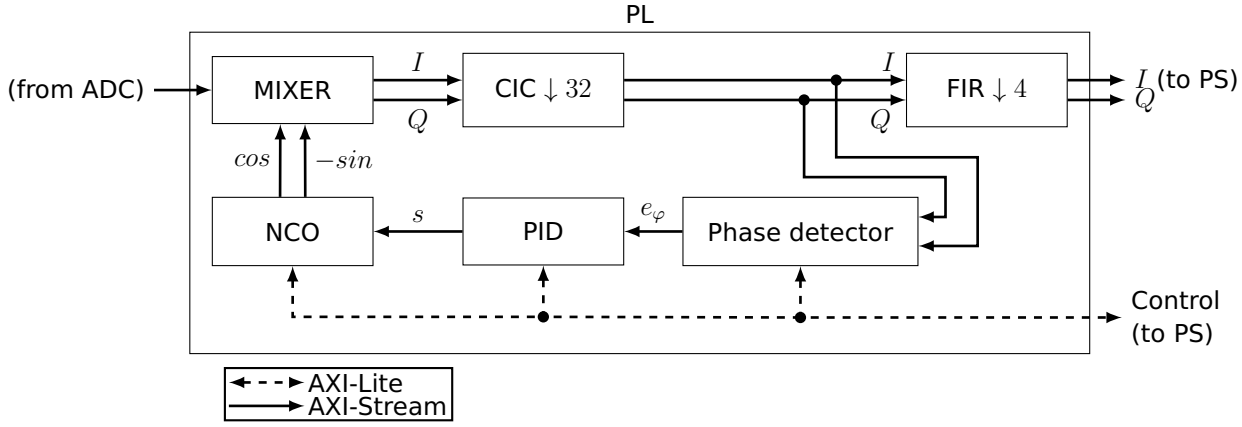

Abbildung 2.1: Architektur des FPGA-Designs (ohne AXI-Infrastruktur)

Die Hauptkomponenten des FPGA-Designs werden in den folgenden Abschnitten näher erklärt.

2.1.1 Mischer (Mixer)

Der Mischer hat die Aufgabe den von dem ADC abgetasteten Datenstrom mit den, vom lokalen Oszillator erzeugten, Trägern zu multiplizieren und so das Nutzsignal in das Basisband zu verschieben. Die besondere Schwierigkeit liegt hier dabei, dass der Mischer mit der Taktrate des ADCs f_{ADC} betrieben werden muss.

Für den Empfang von QPSK-modulierten Daten, ist es notwendig einen komplexen Mischer *(sogenannter I/Q-Mischer)* zu verwenden. Als Eingang dienen zwei vom lokalen Oszillator (NCO) erzeugten Referenzsignale *(Trägersignale)*, sowie der ADC-Datenstrom ($x[n]$), welche dann wie folgt Multipliziert werden:

$$I[n] = x[n] \cdot cos(2\pi \frac{f}{f_{ADC}} \cdot n) \tag{2.1}$$

$$Q[n] = x[n] \cdot -sin(2\pi \frac{f}{f_{ADC}} \cdot n) \tag{2.2}$$

Es entsteht ein komplexer Datenstrom welcher dann für die weitere Verarbeitung genutzt werden kann:

$$\underline{y}[n] = I[n] - j \cdot Q[n] = x[n] \cdot e^{j2\pi \frac{f}{f_{ADC}} \cdot n} \tag{2.3}$$

Durch die reale Natur des Eingangssignales entsteht bei der Mischung, zusätzlich zu dem komplexen Basisbandsignal, auch immer eine Signalkopie bei dem doppelten der Trägerfrequenz. [7] Die nicht erwünschte Kopie, bei doppelter Trägerfrequenz, wird anschließend durch die Verwendung eines Tiefpassfilters eliminiert. Diese Aufgabe übernimmt hier der CIC-Dezimierungsfilter.

Ein- und Ausgabeschnittstellen des Mischers sollen als AXI-Stream Schnittstelle realisiert werden, wobei es notwendig ist jeden Taktzyklus einen neuen Datenwert im Mischer zu verarbeiten.

2.1.2 Lokaler Oszillator (NCO)

Der lokale Oszillator hat die Aufgabe die von dem Mischer verwendeten lokalen Referenzsignale zu erzeugen. Die Frequenz der erzeugten Schwingungen wird hierbei extern, durch Rechenkern oder Phasenregelschleife, vorgegeben.

Umgesetzt ist der lokale Oszillator als Numerisch gesteuerter Oszillator (NCO). Die notwendigen Sinus-/Kosinus-Ausgangssignale werden hierbei durch die sogenannte direkte digitale Synthese (DDS) erzeugt.

Dabei handelt es sich effektiv um einen Zähler (Phasen-Akkumulator, n-Bit breit) von welchem anschließend die untersten m-Bit verwendet werden um die Ausgangsdatenwerte in einer Lookup-Tabelle nachzuschlagen. Anhand der Schrittweite s um welche der Phasen-Akkumulator jeden Taktzyklus erhöht wird lässt sich die Grundfrequenz der ausgegebenen Sinusschwingungen einstellen.

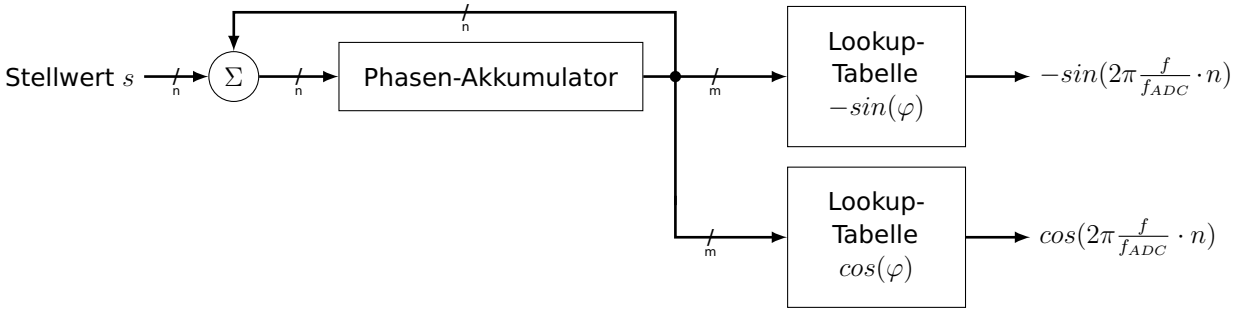

Abbildung 2.2: Schematischer Aufbau des lokalen Oszillators.

Die Breite des Phasen-Akkumulators, die Ausgangswertbreite und die Anzahl der Stützpunkte der Lookup-Tabelle haben einen großen Einfluss auf die Güte der erzeugten Schwingung. [4]

Der Zusammenhang zwischen Frequenzstellwert s und der Ausgangsfrequenz der erzeugten Sinusschwingung $\frac{f}{f_{ADC}}$ ist abhängig von der Akkumulator-Bit-Breite n:

$$\frac{f}{f_{ADC}} = \frac{s}{2^n} \tag{2.4}$$

2.1.3 CIC-Dezimierungsfilter

Nach dem Mischen des Eingangssignales ist es für die weitere Verarbeitung notwendig, die Datenrate des gemischten Signales auf ein beherrschbares Maß zu reduzieren. Vor der Datenreduktion ist es jedoch notwendig die Bandbreite des Signales auf ein adäquates Maß zu beschränken. Zusätzlich dazu ist es notwendig das zweite Mischprodukt, bei doppelter Trägerfrequenz, von dem Nutzsignal im Basisband zu trennen.

Beide diese Aufgaben werden von einem Tiefpassfilter, hier als CIC-Filter ausgeführt, übernommen.

Bei einem Cascaded-Integrator-Comb-Filter handelt es sich grundsätzlich um einen FIR-Mittelwertfilter.

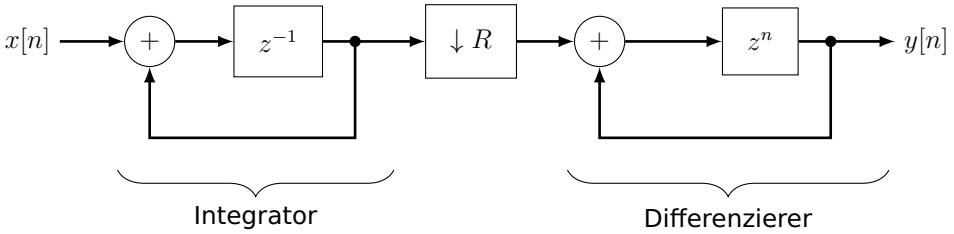

Abbildung 2.3: Struktur eines (transponierten) CIC-Filters 1. Ordnung [8]

Diese Filterstruktur zeichnet sich, besonders bei hohen Taktraten und großen Dezimierungsverhältnissen, jedoch durch eine besonders effiziente Implementierung

aus, da lediglich Subtraktionen und Additionen für die Umsetzung benötigt werden.

Nachteilig hierbei ist jedoch das nur bedingt steuerbare Frequenzverhalten des Filters welche sich grundsätzlich nur durch die Anzahl der Filterstufen (Ordnung N), das Dezimierungsverhältniss R und die Anzahl der Verzögerungsstufen D der Differenzierer steuern lässt. Wobei, anders als bei einem FIR-Filter, nicht jedes beliebige Frequenzverhalten des Filters erreicht werden kann.[8]

Um das erreichen der notwendigen Taktfrequenz während der Implementierung des Filters zu vereinfachen soll der CIC-Filter in der transponierten Form umgesetzt werden. Dies erlaubt die Verkürzung der kritischen Pfade durch die Nutzung der Verzögerungsregister als Pipelining-Register. Es können folglich einfacher höhere Taktraten erreicht werden.

Die notwendigen Filterparameter R und N sollen über generische Parameter während der Erzeugung des FPGA-Bitstreams eingestellt werden können um diese später ohne Änderung des VHDL-Quellcodes anpassen zu können. Zur Vereinfachung der Filterimplementierung soll für D gelten: $D = 1$.

2.1.4 FIR-Kompensationsfilter

Der FIR-Kompensationsfilter hat die Aufgabe der weiteren Reduzierung der Abtastrate bei gleichzeitig notwendiger Bandbreitenbegrenzung. Er wirkt zusätzlich als Kompensationsfilter für den vorgeschalteten CIC-Filter um dessen Schwankungen in der Amplitudenkennlinie im Durchlassbereich zu kompensieren.

Die Filterkoeffizienten wurden mit Matlab anhand des CIC-Filter Frequenzganges und der gewünschten Gesamtfrequenzcharakteristik ermittelt.

Aus Zeit- und Effizienzgründen soll hier der FIR-Filter Compiler[13] von Xilinx verwendet werden.

2.1.5 Phasen-Komparator

Der Phasen-Komparator ermittelt aus den komplexen Basisbandsignalen (I/Q) die Phasenabweichung des lokalen Oszillators bezogen auf das Empfangssignal. Der so ermittelten Phasenfehler kann anschließend verwendet werden um die Frequenz des lokalen Oszillators später so anzupassen, dass er möglichst exakt dem Oszillator des Senders folgt.

Diese Ermittlung des Phasenfehlers erfolgt anhand der in einer Costas-Schleife verwendeten Technik zur Ermittlung des Phasenfehlers.[7] Abhängig von der gewählten Modulationsart (BPSK/QPSK) werden die folgenden Rechenschritte für die Ermittlung des Phasenfehlers genutzt:

Für **BPSK**:

$$e_\varphi = sign(I) \cdot Q \tag{2.5}$$

Für **QPSK**:

$$e_\varphi = sign(I) \cdot Q - sign(Q) \cdot I \tag{2.6}$$

Auf die genaue Herleitung dieser Zusammenhänge soll hier verzichtet werden. Eine genauere analytische Betrachtung [9] könnte später für die Gewinnung eines Modelles der Phasen-Regelschleife genutzt werden. Anhand dieses Streckenmodelles wäre es anschließend möglich den PID-Reglers auszulegen.

2.1.6 PID-Regler

Der PID-Regler wandelt die vom Phasen-Komparator ermittelte Phasenabweichung der Empfangssymbole, in ein Frequenzstellsignal für den lokalen Oszillator um. Dieses Korrektursignal wird anschließend, zusätzlich zu der vorgegeben Oszillatorfrequenz, als Stellsignal für den NCO verwendet. Ziel ist hier die Frequenz- und Phasendifferenz zwischen dem lokalen Oszillator und dem Oszillator des Senders auszuregeln.

Der (im FPGA) zeitdiskret Implementierte Regler setzt eine Übertragungsfunktion zweiter Ordnung um: $F(z) = \frac{A \cdot z^2 + B \cdot z + C}{z^2 - 1}$, deren Parameter A, B, C von außen einstellbar sein sollen. Diese flexible Umsetzung des Reglers soll es ermöglichen den Regler später für unterschiedliche Anwendungsfälle einfach, und zur Laufzeit, anpassen zu können.

Anhand der Bilinear-Transformation (Tustin-Methode) können die zeitdiskreten Reglerparameter aus den Parametern eines zeitkontinuierlich PID-Reglers (K, T_n, T_v) und der Abstastzeit $T_A = \frac{32}{f_{ADC}}$ abgeleitet werden:

$$A = K \cdot (1 + \frac{T_A}{2 \cdot T_n} + \frac{2 \cdot T_v}{T_A}) \tag{2.7}$$

$$B = K \cdot (\frac{T_A}{T_n} + \frac{4 \cdot T_v}{T_A}) \tag{2.8}$$

$$C = K \cdot (\frac{T_A}{2 \cdot T_n} + \frac{2 \cdot T_v}{T_A} - 1) \tag{2.9}$$

Die Implementierung erfolgt anhand der Differenzengleichung welche aus der Übertragungsfunktion abgeleitet werden kann:

$$y[n] = A \cdot x[n] + B \cdot x[n-1] + C \cdot x[n-2] + y[n-2] \tag{2.10}$$

2.2 Software

Nach der Verarbeitung des Empfangssignales soll dieses dem Rechenkern zur weiteren Verwendung überlassen werden. Die Übertragung des Signals zwischen FPGA-Fabric und Rechenkern übernimmt ein DMA-Controller welcher von Xilinx bereitgestellt wird.

Hierzu ist es notwendig Software für den Rechenkern zu erstellen, die sowohl die erzeugten Nutzdaten entgegennimmt als auch die Steuerung des FPGA-Designs übernimmt.

Um die Nutzung des SDRs auch bei wenig Erfahrung mit der Erstellung von Software für Eingebettete Systeme zu ermöglichen, soll PYNQ[1] als Softwareframework zum Einsatz kommen.

Dabei handelt es sich um ein Linux-Image mit vorinstallierter Software welche die Nutzung von FPGA-Resourcen aus der Programmiersprache Python ermöglicht. Zusätzlich kommt eine Web-basierten Oberfläche (Jupyter) zum Einsatz welche es erlaubt die notwendigen Python-Programme über das Netzwerk und ohne notwendige Software zu erstellen.

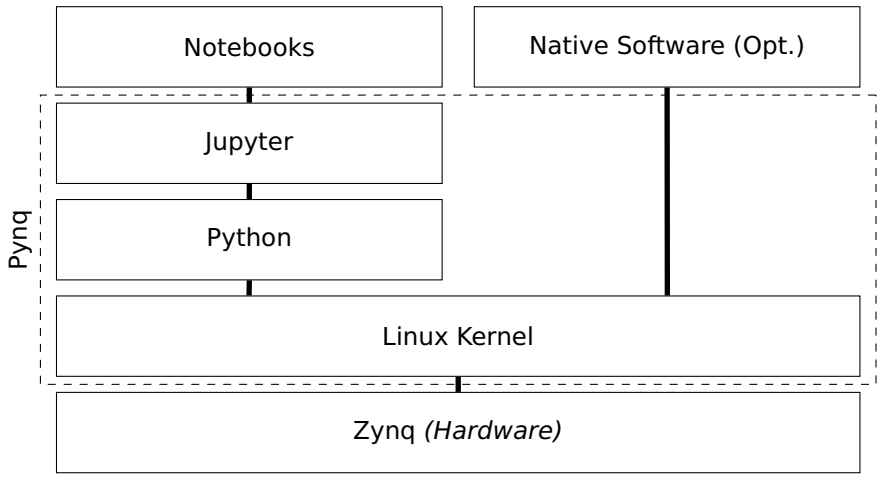

Abbildung 2.4: Grundlegender Aufbau der Software

Der Hauptteil der Software wird hier in der Form von sogenannten Jupyter-Notebooks erstellt. Dabei handelt es sich grundsätzlich nur um einfache Python Programme die jedoch in einer Weboberfläche erstellt werden und zusätzlich Textuelle Beschreibungselement enthalten können. Diese Notebooks werden anschließend durch den systemeigenen Python-Interpreter auf dem Rechenkern *(Processing System)* ausgeführt.

Hierbei ist weder großes Linux/FPGA oder C-Wissen notwendig um mit den im FPGA enthaltenen Komponenten zu Interagieren.

Für performance-kritische Anwendungen wäre es jedoch auch möglich native Software zu erstellen. Diese könnte z.B. mit VITIS erstellt und anschließend entweder aus Python aufgerufen oder Eigenständig auf dem System ausgeführt werden.

Die Interaktion zwischen der Software und den Komponenten des FPGA-Designs erfolgt über den Zugriff auf Special Function Registers auf welche als Memory-Mapped-IO-Device direkt oder über Python Bibliotheken zugegriffen werde kann. Die Linux spezifischen Facetten dieses Hardwarezugriffs (z.B. Gerätetreiber und Device-Tree-Overlays) werden hierbei von der PYNQ Implementierung übernommen.

[1] http://www.pynq.io

Für das verwendete Board Eclypse-Z7 wird kein fertiges PYNQ Image angeboten. Es ist also notwendig ein eigenes Image anhand der vorhandenen Board-Konfiguration zu erstellen. Hierzu wird eine - mehr oder weniger - umfangreiche Anleitung zur Verfügung gestellt. [2]

3 Implementierung

Nach dem Entwurf des Systems wurde mit der Implementierung begonnen. Hierbei wurde zuerst das FPGA-Design umgesetzt und anschließend durch Simulation verifiziert.

Nachfolgend wurde mit der Erstellung des PYNQ-Images begonnen und erst abschließend wurde die Python-Software(Notebooks) zur Systemsteuerung umgesetzt. Final erfolgte ein Test des Gesamtsystems durch Empfang eines, durch einen Funktionsgenerator erzeugten, Testsignales.

Die folgenden Abschnitte sollen detailliert den verwendeten Entwicklungsprozess sowie das vorgehen bei der Implementierung der einzelnen Komponenten beschreiben.

3.1 FPGA-Design

Die Erstellung des FPGA-Designs erfolgte in der Hardwarebeschreibungssprache VHDL. Diese wurde aufgrund von bestehender Erfahrungen und der bekannten Syntax ausgewählt.

Als Entwicklungsumgebung für die in VHDL geschriebenen Komponenten wurde die **Sigasi IDE**[1] verwendet. Hierbei handelt es sich um eine Eclipse basierte Entwicklungsumgebung, die sich (im Vergleich zu Proprietären FPGA-Entwicklungsumgebungen) besonders durch ihre einfache Bedienung und eine schnelle Reaktionszeit auszeichnet.

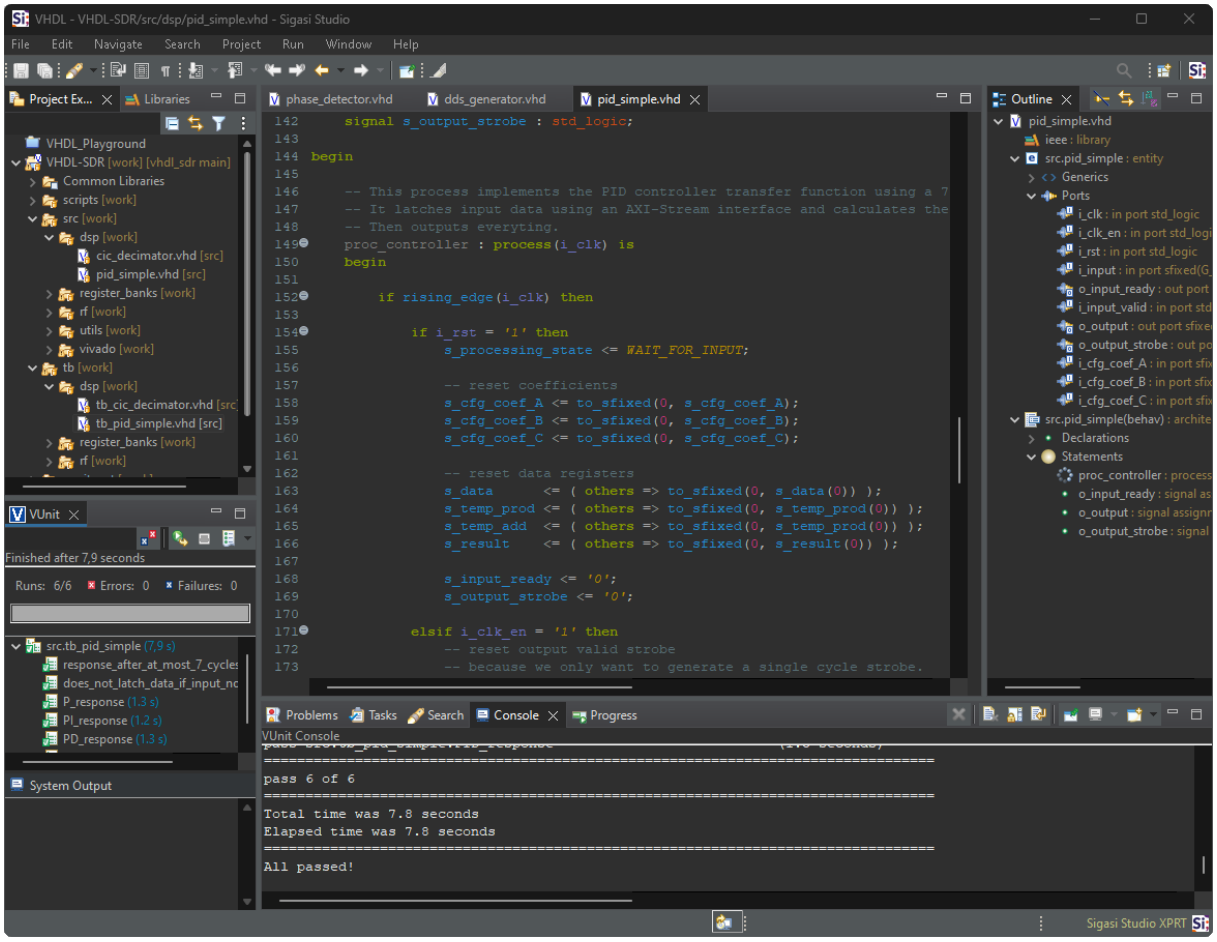

Abbildung 3.1: Screenshot der Sigasi IDE

Besonders Komfortabel an dieser Entwicklungsumgebung, ist die Möglichkeit externe Simulatoren (z.B. XSIM oder GHDL) sowie externe Testframeworks (z.B. OSVVM, VUnit) einfach integrieren zu können und so Tests direkt aus der Entwicklungsumgebung möglich machen.

[1] https://www.sigasi.com

Als Verifikationsframework wurde **VUnit**[2] verwendet. Hierbei handelt es sich um Open-Source Unit-Testing Framework für VHDL welches über Python angesteuert werden kann und bereits viele Verfikationskomponenten (z.B. Bus-Master) beinhaltet.

Als besonders nützlich stellte sich hier die Möglichkeit Python für die Erstellung und Verifikation von Testdaten nutzen zu können heraus. Das hierzu angewendeten Verfahren wird näher bei der Verifikation der einzelnen Komponenten beschrieben.

Als Simulator wurde **GHDL**[3] verwendet.

Für die Synthese und OnChip-Implementierung des Designs sowie die Erstellung des Block-Designs des FPGA-Bitstreams wurde die von Xilinx bereitgestellte Entwicklungsumgebung **Vivado** verwendet.

3.1.1 Vorgehen bei der Implementierung

Begonnen wurde die Implementierung mit der Erstellung des VHDL-Codes.

Hierbei wurden zuerst die Eingangs-/Ausgangsports der einzelnen Komponenten spezifiziert. Anschließend wurde die im Entwurf festgelegte Funktionalität umgesetzt.

Wo möglich wurden funktional zusammengehörige Gruppen in eigene Komponenten ausgelagert, diese dann separat umgesetzt und dabei möglichst generisch gestaltet.

Besonderes Augenmerk lag ebenfalls auf dem effizienten Mapping von VHDL-Code auf die einzelnen primitiven Logikelementen welche der FPGA zur Verfügung stellt.. So wurde z.B. besonders Multiplikationen so umgesetzt, dass bei der späteren Synthese möglichst viele vorhandene DSP-Slices bei der Synthese verwendet werden können. Besonders hilfreich war hier die von Xilinx zur Verfügung gestellte Dokumentation zum Aufbau der vorhanden DSP-Slices[10].

Nach bzw. während der Erstellung des VHDL-Codes wurde eine funktionelle Simulation der einzelnen Komponenten durchgeführt. Hierbei wurden Testfälle (Testbenches) erstellt welche, soweit möglich, die Funktionalität der Komponenten bereits selbstständig verifizieren. Dies sollte den manuelle Verifikationsaufwand gering halten und die Wiederholbarkeit der Testfälle erhöhen. Besonders vereinfacht wurde die Testfallerstellung durch die von VUnit bereitgestellten Bibliotheken sowie der Möglichkeit externe Python-Scripte in den Testfällen zu verwenden. Wo eine automatische Testfallüberprüfung nicht möglich wurde das Verhalten der Komponenten manuell überprüft.

Nach Durchführung der Simulation wurden die einzelnen Komponenten separat in Vivado synthetisiert. Anhand diesen synthetisierten Designs wurden anschließend

[2] https://vunit.github.io
[3] http://ghdl.free.fr

das Erreichen der gewünschten maximalen Taktrate überprüft. Wurde diese initial nicht erreicht so wurde der VHDL-Code durch Einführung von Zwischenregistern (Pipelining) angepasst, bis die gewünschte Taktrate erreicht wurde.

Besonders wichtig, wenn auch mühselig, war hier ebenfalls die Kontrolle des von Vivado erstellten Schaltplanes um zu überprüfen ob die gewünschten Hardwareprimitiven (z.B. DSP-slices) richtig eingefügt werden konnten. Hilfreich waren hier besonders die von Vivado bereitgestellten Build-Logs aus welchen Informationen zu verwendeten primitiven gewonnen werden konnten (DSP-Utilization-Report).

Nach funktionaler Fertigstellung des VHDL-Codes wurde abschießend Dokumentation erstellt welche die einzelnen Komponenten sowie deren interne Funktion beschreiben und erklären soll. Die Dokumentation wurde in der Form von Inline-Code-Dokumentation erstellt.

Der hier beschriebenen Implementierungsprozess wird ebenfalls von 3.2 dargestellt.

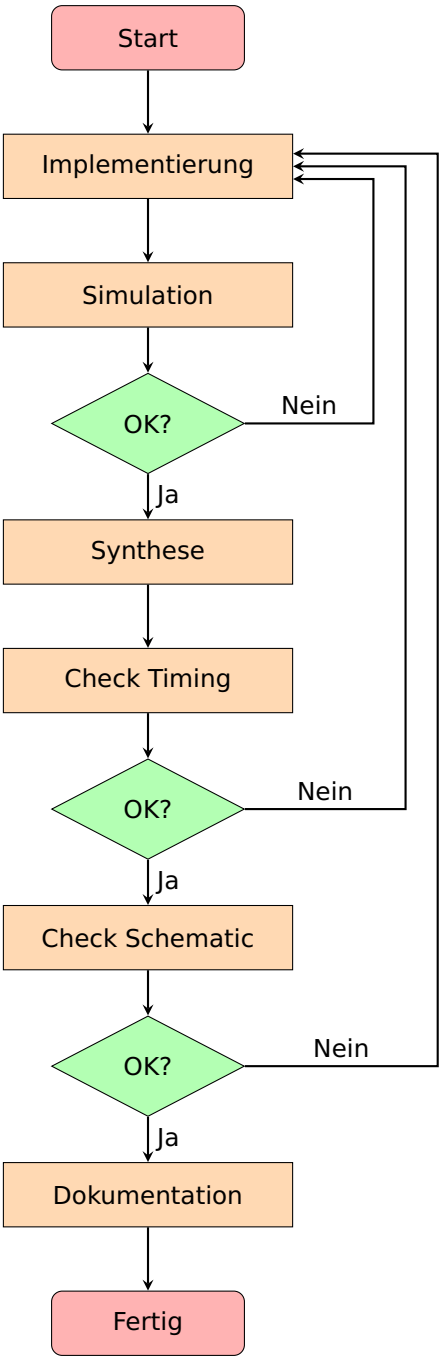

Abbildung 3.2: Ablauf der Implementierung

3.1.2 Mischer (Mixer)

Der Mischer hat die Aufgabe den vom ADC gewonnenen Datenstrom mit den vom NCO erzeugten Trägersignalen zu Multiplizieren. Aufgrund der Tatsache, dass dies vor der Dezimierung des Eingangssignals erfolgt, handelt es sich hier um eine kritische Stelle. Deren Performance die maximale Taktrate bestimmt, mit dem das System betrieben werden kann.

Es wurde deswegen besonders darauf geachtet, den VHDL-Code so zu gestalten, dass für die notwendigen Multiplikationen DSP-Slices verwendet werden können.

Die Multiplikationsoperanden sowie das Ergebnis werden im Festkomma-Format dargestellt. Das Ergebnis der Multiplikation wird auf die Bit-Breite des Eingangssignales gerundet. Die Rundungsart ist einstellbar.

3.1.3 Lokaler Oszillator (NCO)

Die Umsetzung des lokalen Oszillators erfolgt anhand der in 2.1.2 beschriebenen Struktur. Da hier ebenfalls einmal pro ADC-Taktzyklus Ausgangswerte generiert werden müssen, handelt es sich hier ebenfalls um ein Performance-kritisches Modul.

Es wurde deswegen besonders drauf geachtet, die notwendigen Sinus/Kosinus Lookup-Tabellen als Block-RAM (BRAM) umsetzen zu können.

Für langsame Trägersignale wurde hier zusätzlich die (optionale) Möglichkeit geschaffen zwischen Stützstellen der Lookup-Tabelle linear zu interpolieren. Diese Option bietet besonders bei geringen Trägerfrequenzen und einer geringen Anzahl von Stützstellen einen Gewinn in der Güte des Trägersignales. Dies erfolgt jedoch auf Kosten der maximalen erreichbaren Taktfrequenzen.

Zusätzlich kann jeder Träger optional invertiert werden um etwaige Vorzeichenprobleme im Empfänger auszugleichen.

Nach [4] wären mehrere Maßnahmen denkbar, welche die Größe der Lookup-Tabelle verringern ohne die Anzahl der Stützpunkte zu reduzieren. Aus Einfachheit wurde hier jedoch drauf verzichtet und der Oszillator verwendet eine Lookup-Tabelle deren Größe der Anzahl der Stützpunkte entspricht.

Die unterschiedlichen Parameter des Oszillators (Akkumulatorbreite n, Anzahl der Stützstellen 2^m, sowie die Bitbreite der erzeugten Träger kann über generische Parameter während der Synthese eingestellt werden.

Die Verifikation des Oszillators erfolgte durch eine spektrale Analyse der erzeugten Träger. Anhand des ermittelten Spektrums der erzeugten Träger wurden Position der Grundschwingung, Anteil und Amplitude der Harmonischen und die Phasenlage der Träger zueinander ermittelt. Dies erfolgte für unterschiedliche Parameterkonfigurationen.

Die folgende Abbildung 3.3 zeigt beispielhaft das Spektrum eines erzeugten Kosinus-Trägers für eine Oszillatorfrequenz $f = 14{,}41\,\mathrm{MHz}$ bei einer ADC-Taktrate von $f_{ADC} =$

$100\,\mathrm{MHz}$ einer Akkumulatorbreite von $n = 21\,\mathrm{bit}$ und einer Anzahl von 1024 Stützpunkten ($m = 10\,\mathrm{bit}$)

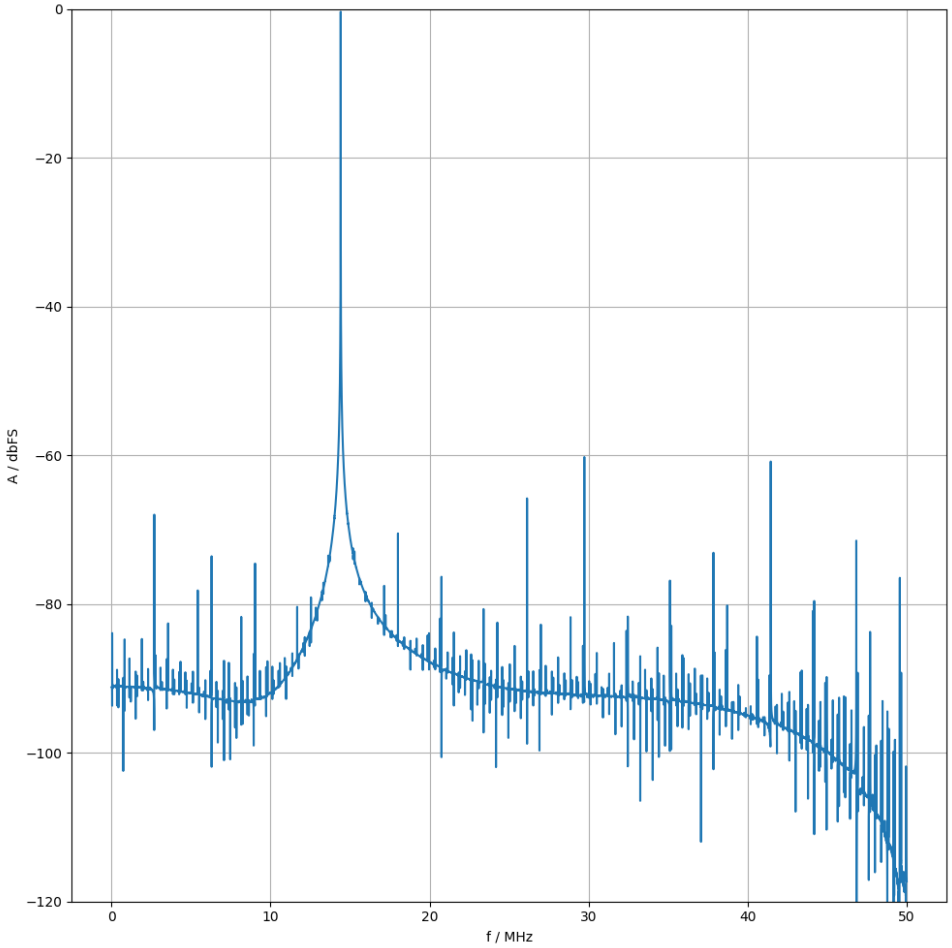

Abbildung 3.3: Spektrum des cos Trägers für: $f_{adc} = 100\,\mathrm{MHz}, f = 14{,}41\,\mathrm{MHz}, n = 21, m = 10$

Die aus der Theorie erwarteten Kenngrößen[4] konnten durch die gemessen Daten bestätigt werden.

3.1.4 CIC-Dezimerungssfilter

Die Implementierung des CIC-Dezimierungssfilter wurde anhand des in 2.1.3 spezifizierten Entwurfs durchgeführt.

Soweit möglich können die Filterparameter (Ordnung N, Dezimierungsrate R) über generische Parameter während der Synthese festgelegt werden.

Um die Implementierung des Filters zu vereinfachen wurde hier jedoch bestimmte Einschränkungen der Parameter getroffen:

- der Dezimierungsfaktor muss eine Zweierpotenz sein
  (Dies vereinfacht die Normalisierung des Filter-Gains, da die Division durch Potenzen von Zwei durch Routing erfolgen kann)
- die Filterordnung muss größer oder gleich dem Dezimierungsfaktor sein
  (Notwendig um die Differenzierer, sequenziell und ohne Pipelining, umsetzen zu können)

Da der CIC-Filter, vor der Dezimierung, ebenfalls Daten mit dem ADC-Taktfrequenz verarbeitet, handelt es sich hier ebenfalls um eine Komponente mit kritischen Pfaden. Optimierungspotential wäre hier, durch Flexibilisierung der intern verwendeten Bit-Breiten (pruned-CIC-Filter) und folglicher Reduktion der Schaltungskomplexität, noch vorhanden.

Die Verifikation des Filters erfolgte durch Vergleich der Sprungantwort mit einer von Matlab erzeugten Referenzkurve.

3.1.5 Phasen-Komparator

Der Phasen-Komparator ermittelt anhand der im Entwurf (vergleiche 2.1.5) beschriebenen Gleichungen die Phasenabweichung des Eingangssignals.

Da der Phasen-Komparator bereits mit dem dezimierten Eingangssignal arbeitet sind hier mehrere Taktzyklen für die Verarbeitung des Datenwerts vorhanden. Die Durchführung der Berechnung wurde deswegen in der Form eines Zustandsautomates umgesetzt.

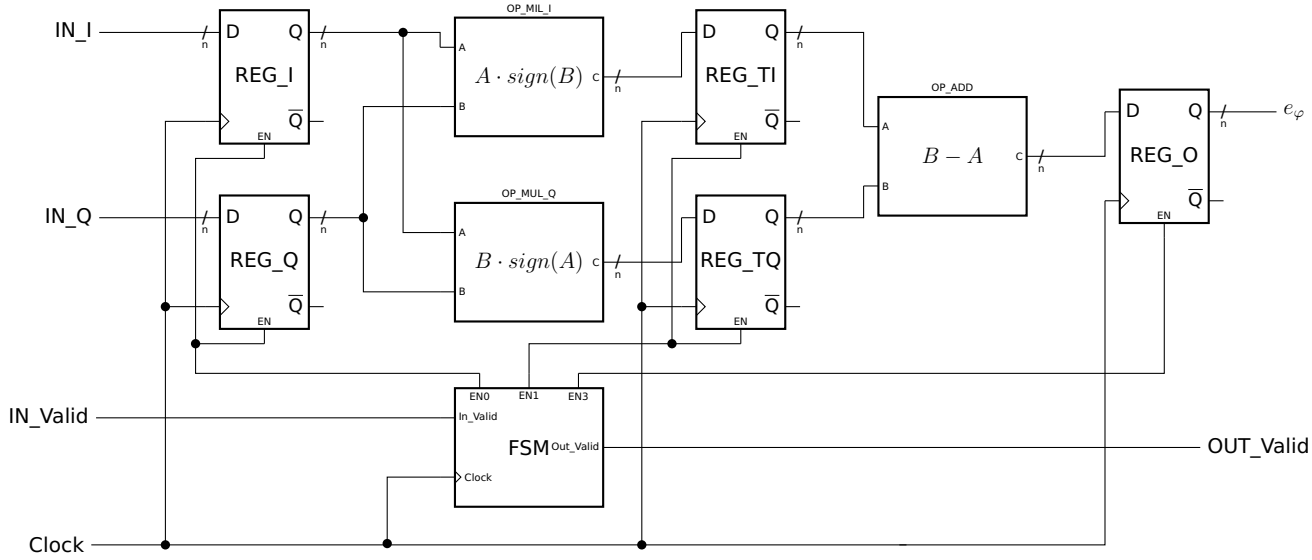

Abbildung 3.4: Vereinfachter Schaltplan des Phasen-Komparators.

Ein Zustandsautomat übernimmt hier die Erzeugung der Clock-Enable Signale welche den Datenfluss durch das Modul steuern. Die explizite Datenflusssteuerung ist notwendig, da nicht jeden Taktzyklus gültige Daten am Eingang empfangenen werden. Da die Ermittlung des Phasenfehlers bereits mit reduzierter Datenrate erfolgt,

ist es darüber hinaus nicht notwendig jeden Taktzyklus einen Wert verarbeiten zu können.

Zwischen zwei Registern befindet sich die kombinatorische Logik welche die notwendigen arithmetischen Operationen umsetzt und so die eigentliche Datenverarbeitung übernimmt.

Die Umschaltung zwischen den Modulationsarten, im obigen Schaltplan aus Übersichtlichkeitsgründen nicht enthalten, erfolgt durch Anpassung der Differenzbildung im Block `OP_ADD`, für BPSK: $C = B$, für QPSK: $C = B - A$.

3.1.6 PID-Regler

Der PID-Regler implementiert die in Absatz 2.1.6 ermittelte Übertragungsfunktion.

Die Umsetzung erfolgt hier, ähnlich wie bei dem Phasen-Komparator, wieder anhand eines Zustandsautomaten der die einzelnen Rechenoperationen sequenziell abarbeitet.

Es werden dadurch mehrere Taktzyklen für die Verarbeitung eines Eingangswertes benötigt. Da der PID-Regler jedoch ebenfalls bereits dezimierte Signale verarbeitet stehen diese Taktzyklen zur Verfügung.

Die einzelnen für die Berechnung notwendigen Rechenschritten wurden so umgesetzt, mit so eventuell notwendigen Wartezyklen, so das möglichst viele Operation (Multiplikationen/Additionen) in DSP-Slices verlagert werden können. So ist lediglich ein mit Logikelementen umgesetzter Addierer für das Endergebnis nötig.

Die Berechnung der Reglerantwort wird intern jeweils in Festkommadarstellung und mit voller Genauigkeit durchgeführt. Am Ende der Berechnung erfolgt eine einzelne Rundung auf das Ausgangsformat. Die Rundungsart (Runden,Abscheiden) ist hierbei einstellbar.

Die Verifikation erfolgte durch Überprüfung der Sprungantwort für unterschiedliche Werte der Reglerparameter A, B und C. Die hierzu verwendete Referenzsprungantwort wurde mit Matlab erzeugt.

3.1.7 RF-Reciever

Der RF-Reciever bildet das Top-Level Modul des in VHDL erstellten Teils des FPGA-Designs.

der RF-Reciever verschaltet Mischer, den lokalen Oszillator, die CIC-Dezimierungsfilter für I und Q Datenpfad, den Phasen-Komparator und den PID-Regler zu einer Phasen-Regelschleife welche die Demodulation des Eingangssignales inklusive Trägerrekonstruktion übernimmt.

Grundsätzlich erfolgt hier größtenteils nur eine Instanziierung der bereits erstellten Komponenten. Lediglich kleinere Logikteile, welche benötigt werden um die Komponenten verschalten zu können, sind hier umgesetzt.

So ist z.B. zwischen lokalem Oszillator und PID-Regler ein Register notwendig, welches die AXI-Stream Schnittstelle am Ausgang des PID-Reglers auf den direkten Logikeingang des Oszillators anpasst.

Zusätzlich wurde der gesamte Empfangsteil anschließend, für unterschiedlichen Parameter (Trägerfrequenzen, Signal/Rauschverhältnis), durch Simulation getestet. Es wurde zusätzlich die empirische Einstellung des Reglers vorgenommen.

Bei der Simulation wurden, mit Python, jeweils ein modulierter Datenstrom erzeugt der, anstatt den ADC-Daten, in das System eingespeist wurde. Hierbei wurden Trägerfrequenz f_c, Signal/Rausch-Verhältnis SNR, Frequenzfehler des Träges Δf, und Signalamplitude A variiert. Der demodulierte Datenstrom I/Q wurde anschließend mit dem gesendeten Datenstrom verglichen.

Ein Simulationsergebnis, bestehend aus demodulierten Daten in komplexer und polarer Darstellung sowie den Stellwert der Phasen-Regelschleife kann im Anhang 5.4 gefunden werden.

3.1.8 Generische AXI-Lite Registerbank

Um die Erstellung der AXI-Lite Registerbänke zu vereinfachen, wurde eine generische AXI-Lite Registerbank erstellt.

Diese wandelt alle, auf dem AXI-Lite Bus empfangenen Lese und Schreibtransaktionen in ein einfacheres Protokoll, welches dann zur Umsetzung von konkreten Registerbänken verwendet werden kann.

Auf eine genaue Analyse des AXI-Lite Protokolls soll hier verzichtet werden.

Grundlegend lässt sich jedoch zusammenfassen, dass ein AXI-Lite Bus aus unterschiedlichen Kanälen (Channels) besteht die jeweils über einen Ready/Valid-Handshake Nutzdaten übertragen.
Das Ready Signal gibt hierbei an ob ein Geräte bereit ist Daten zu empfangen. Das `Valid` Signal wird genutzt um zu Signalisieren, dass gültige Daten auf dem Bus anliegen. Ein Transfer gilt als abgeschlossen wenn gilt: `Ready = Valid = 1`.[3]

Grundbestandteile der generischen Registerbank sind zwei Zustandsautomaten, wobei jeweils ein Automat die Lesezugriffe und ein Automat die Schreibzugriffe behandelt. Die für die Transaktionen jeweils benötigten AXI-Lite Channels werden dabei jeweils von einem der Automaten verarbeitet.

Die „Read FSM“ wandelt dabei die AXI-Lite read data und address Kanäle in ein einfaches kombinatorisches Interface um.

Von dem untergeordneten Modul wird erwartet, dass einen `Clock`-Zyklus nachdem eine Adresse via `R_Address` bereitgestellt wurde das Ergebnis der Leseoperation via

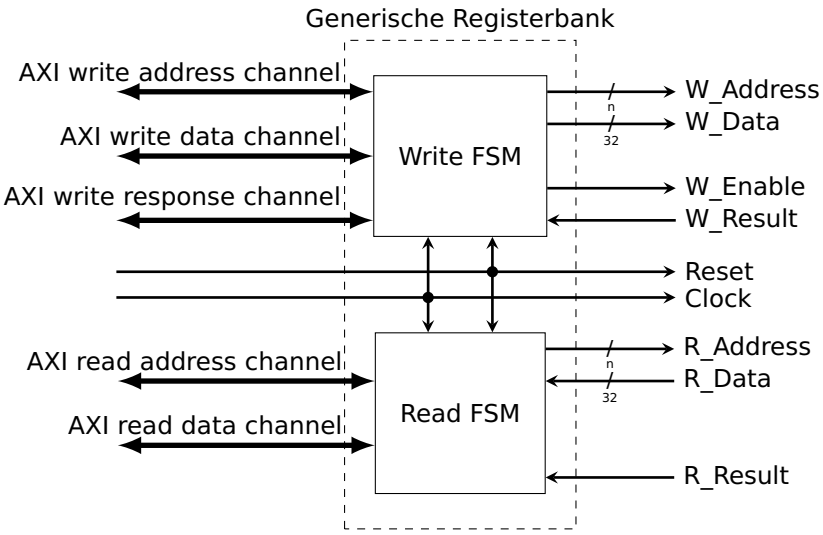

Abbildung 3.5: Schematischer Aufbau der generischen AXI-Lite Registerbank.

`R_Data` bereitgestellt wird.
`R_Result` kann hierbei verwendet werden um ungültige Lesezugriffe abzulehnen.

Die „Write FSM“ wandelt die AXI-Lite write address, data und response Kanäle in ein einen synchronen write-enable basierten Bus um.

Von dem untergeordneten Modul wird hierbei erwartet, dass es bei jeder steigenden Flanke von `Clock`, solange `W_Enable` aktiv ist, die in `W_Data` bereitgestellten Daten in das Register an der Adresse `W_Address` speichert.
`W_Result` kann hierbei verwendet werden um ungültige Schreibzugriffe abzulehnen.

3.1.9 Block-Design

Das Block Design, welches gleichzeitig das Toplevel-Design des FPGAs darstellt, wurde mit Vivado erstellt. Hierbei wurden die in VHDL erstellten Komponenten als RTL-Blöcke in das Block-Design übernommen und anschließend mit den weiteren notwendigen Komponenten verbunden.

Hierbei wurden auch alle zusätzlich, für die Interaktion der RTL-Komponenten untereinander oder mit dem Rechenkern, benötigten AXI-Infrastruktur Blöcke eingefügt. Notwendige zusätzlich Blöcke waren:

1. **AXI DMA-Controller**: für die Übertragung der AXI-Stream Daten an die CPU[11].
2. **AXI-Interconnect**: für die Verbindung der AXI-Busse welche für Steuerregister oder die DMA-Controller verwendet werden[12].
3. **AXI-Stream FIFO/Converter**: für die Verbindung der einzelnen RTL Module untereinander und mit dem DMA-Controller wenn Daten umgewandelt oder gepuffert werden mussten[14].

Zusätzlich wurde der von Digilent bereitgestellte ZModScopeController[6] verwendet um mit dem ADC zu kommunizieren.

Dabei ist zu beachten, dass es bei Vivado nur möglich ist RTL-Komponenten in das Blockdesign zu integrieren, welche in VHDL-2000 oder niedriger erstellt wurden und `std_logic` basierte Ein/Ausgänge benutzen.

Da der Großteil des VHDL-Codes in VHDL-2008 erstellt wurde und zusätzlich viele selbst definierte Typen für Ein/Ausgabeports verwendet wurden, war es notwendig Wrapper der einzelnen Komponenten zu erstellen. Diese abstrahieren alle nicht `std_logic` basierte Datentypen und passten das Sprachlevel auf VHDL-2000 an. Dies erfolgte üblicherweise in zwei Schritten, ein VHDL-2008 basierter Wrapper welche alle Datentypen in `std_logic` oder `std_logic_vector` umwandelt. Und anschließend ein zweiter Wrapper, welcher das Sprachniveau auf VHDL-2000 bringt.
Im Quellcode wurden Wrapper nach folgendem Schema benannt:
`<name_des_moduls>_wrapper_<sprach_level>.vhd`

Eine Übersicht über das Blockdesign kann im Anhang 5.1 gefunden werden.

3.2 Software

3.2.1 PYNQ-Image

Vor der Erstellung der Software war es notwendig das PYNQ-Image für den FPGA zu erstellen. Hierzu wird ein FPGA-Design benötigt welches die Standardkonfiguration der Rechenkerne beschreibt. Als FPGA-Design wurde das bereits erstellte Design verwendet.

Anhand des FPGA-Designs ist es dann möglich ein PYNQ Board-Support-Package zu erstellen. Dieses beinhaltet die gesamte Board spezifische Konfiguration. Um dieses Board-Support-Package zu erstellen wurde die PYNQ Dokumentation befolgt [2].

Anhand des Board-Support-Packages ist es anschließend möglich das PYNQ-Image zu erstellen.

Es wurde die Ubuntu version 22.04 verwendet, welche von PYNQ nicht direkt unterstützt wird. Es war deshalb notwendig die PYNQ-Scripte so anzupassen, dass Überprüfungen ob die korrekte Betriebssystemversion verwendet wurde auch für Ubuntu 22.04 erfolgreich waren.

Zusätzliches Hindernis war das in der verwendeten Ubuntu-Distribution standardmäßig ZFS mit Kompression zu Einsatz kam. Dies wird von PYNQ nicht unterstützt und es kommt zu einem Fehler bei der Erstellung des SD-Karten Images. Es ist deswegen notwendig die ZFS-Kompression für den Ordner, aus dem das PYNQ-Buildsystem aufgerufen wird, zu deaktivieren.

Nach der Erfolgreichen Erstellung des PYNQ-Images konnte dieses auf eine SD-Karte gespielt und auf dem Board getestet werden.

3.2.2 Jupyter-Notebooks

Die Erstellung der eigentlichen und nur sehr rudimentär ausgeprägten Steuersoftware wurde anschließend über das Jupyter Webinterface vorgenommen.

Dazu wurde ein Jupyter Notebook erstellt welches:

1. den FPGA-Bitstream auf den PL-Teil des FPGAs läd
2. die Steuerregister für ADC und RF-Demodulator mit den korrekten Werten beschreibt
3. den DMA-Controller in Betrieb nimmt, so dass Daten zum Rechenkern übertragen werden können
4. die vom DMA-Controller empfangenen Daten anzeigt

Hierbei war vor allem die Dokumentation und der Quellcode der von PYNQ bereitgestellten Python Bibliothek sehr hilfreich[1].

4 Fazit

Die konzeptionelle als auch technische Umsetzung des Projektes war sehr Lehrreich.

Besonders in dem Bereiche der digitalen Signalverarbeitung (Filter-Design und deren Umsetzung im FPGA) als auch im Bereich der Empfängerarchitekturen konnten viele neue Erkenntnisse gewonnen werden. Ebenfalls sehr interessant war die Erstellung des PYNQ-Images und die damit verbundene Arbeit mit dem Linux-Kernel.

Alles in allem, erfüllt das Projekt die anfangs gestellten Erwartungen, auch wenn er tatsächliche Empfang von Funksignalen nur mäßig funktioniert. Hauptproblem hierbei ist wohl die korrekte Auslegung des PID-Reglers. (vergleiche: 4.1.1)

Nach einigem Rumprobieren mit den Reglerparametern ist der Empfang von QPSK-Modulierten Signalen zwar möglich, jedoch nur bei „schnellen“ Symbolraten und ohne Pulse-Shaping.

4.1 Verbesserungspotential

Aufgrund der in Summe deutlich überschrittenen Projektdauer wurden an einigen Stellen Vereinfachungen getroffen die jedoch bei einer Weiterbearbeitung des Projektes behoben werden sollte. Der folgende Abschnitt soll einige Möglichkeiten der Verbesserung des Projektes aufzeigen.

4.1.1 Auslegung des PID-Reglers

Die Auslegung der Reglerkoeffizienten erfolgte Experimentell, in der Simulation oder auf dem FPGA bei einem konstanten Eingangssignal. Die so ermittelten Koeffizienten sind jedoch nicht optimal. Sowohl bezüglich auf die Stabilität des Regelkreises als auch in der erreichten Einschwingzeit wären durchaus noch Optimierungen denkbar.

Eine Verbesserung hier wäre die Erstellung eines möglichst akkuraten Streckenmodelles, anhand dessen anschließend die Reglereinstellung nach mathematischen Prinzipien erfolgen könnte, z.B. durch Polstellenplatzierung.

Schwierigkeit hier stellt die Erstellung des Streckenmodelles dar. Zusätzliche Komplexität folgt aus der Tatsache, dass die Regelkreisauslegung im Zeit-diskreten erfolgen müsste um die vorhandenen Totzeiten des Systemes berücksichtigen zu können.

4.1.2 Ergänzung der Phasen-Regelschleifen für FSK

Bei korrekter Einstellung der Phasen-Regelschleife wäre es möglich diese für den Empfang von FSK-Signalen zu nutzen.

Hierbei wäre es möglich den Stellwert des Phasen-Reglers als Ausgangssignal zu nutzen um so FSK modulierte Daten zu dekodieren. Damit dies möglich wäre, ist es nötig diesen Stellwert dem Rechenkern zur Verfügung zu stellen. Dies könnte wohl am einfachsten erfolgen, indem die AXI-Stream basierte Ausgangsschnittstelle des Reglers dupliziert und einem weiteren DMA zugeführt wird.

4.1.3 Flexibilisierung des FIR-Filters

Der aktuell verwendete FIR-Kompensationsfilter bietet, im Bezug auf die Flexibilität wenig Optionen, da er nicht Re-konfigurierbar ist. Sowohl die Dezimierungsrate als auch die Filterkoeffizienten werden aktuell während der Synthese fixiert.

Die bereits in dem IP-Core vorhandene Konfigurationsschnittstelle könnte genutzt werden um die Koeffizienten des Filters zur Laufzeit zu ändern. Hierzu wäre es notwendig eine AXI-Lite Registerbank zu erstellen, welche die Re-konfiguration des Filters, anhand der vom Rechenkern erhaltenen Koeffizienten, übernimmt.

Die Dezimierung könnte ebenfalls verbessert werden, indem diese im Nachgang des Filters erfolgt. Hierbei wäre es ebenfalls möglich eine AXI-Lite basierte Registerbank für die Konfiguration zu nutzen.

4.1.4 Inbetriebnahme des zweiten ADC-Kanals

Im aktuellen Design wird nur ein Kanal des ADCs verwendet.

Durch hinzufügen eines zweiten RF-Reciever IP-Blocks wäre es möglich auch den zweiten Kanal des ADCs nutzen zu können. Zusätzlich wäre es nötig, die notwendige AXI-Infrastruktur anzupassen und weitere DMA-Controller hinzuzufügen.

Die Schnittstelle für die ADC-Konfiguration unterstützt bereits zwei Kanäle.

Literaturverzeichnis

[1] Advanced Micro Devices, Inc. pynq package. https://pynq.readthedocs.io/en/latest/pynq_package.html. [Online; Abgerufen am: 03.03.2024].

[2] Advanced Micro Devices, Inc. Pynq sd card image. https://pynq.readthedocs.io/en/latest/pynq_sd_card.html. [Online; Abgerufen am: 28.02.2024].

[3] ARM, Inc. *AMBA AXI and ACE Protocol Specification - AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite (ID102711)*, 2011. v1.10.

[4] L. Cordesses. Direct digital synthesis: a tool for periodic wave generation (part 1). *IEEE Signal Processing Magazine*, 21(4):50–54, 2004.

[5] Digilent Inc. Eclypse-z7 reference manual. https://digilent.com/reference/programmable-logic/eclypse-z7/reference-manual. [Online; Abgerufen am: 22.02.2024].

[6] Digilent Inc. Zmod scope reference. https://digilent.com/reference/zmod/scope/start. [Online; Abgerufen am: 22.02.2024].

[7] Qasim Chaudhari. Costas loop for carrier phase synchronization. https://wirelesspi.com/costas-loop-for-carrier-phase-synchronization/. [Online; Abgerufen am: 27.02.2024].

[8] Rick Lyons. A beginner's guide to cascaded integrator-comb (cic) filters. https://www.dsprelated.com/showarticle/1337.php. [Online; Abgerufen am: 27.02.2024].

[9] Xiao-ou Song. Analysis and implementation of modified costas loop for qpsk. In *2015 International Conference on Intelligent Networking and Collaborative Systems*, pages 394–397, 2015.

[10] Xilinx, Inc. *User Guide - 7 Series DSP48E1 Slice (UG479)*, 03 2018. v1.10.

[11] Xilinx, Inc. *AXI DMA v7.1 LogiCORE IP Product Guide (PG021)*, 04 2022.

[12] Xilinx, Inc. *AXI Interconnect v2.1 LogiCORE IP Product Guide (PG059)*, 05 2022.

[13] Xilinx, Inc. *FIR Compiler v7.2 LogiCORE IP Product Guide (PG149)*, 10 2022.

[14] Xilinx, Inc. *AXI4-Stream Infrastructure IP Suite v3.0 LogiCORE IP Product Guide (PG085)*, 05 2023.

5 Anhang

5.1 Block-Design

Blockdesign: Top-Level

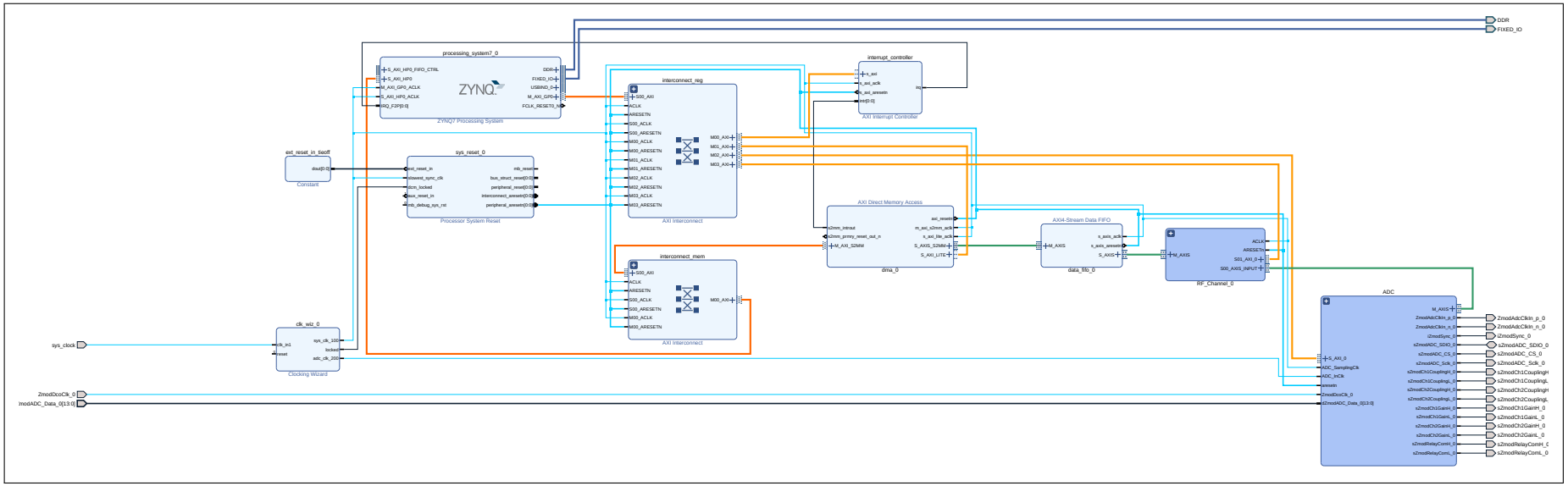

Abbildung 5.1: Block-Design des Top-Level Moduls

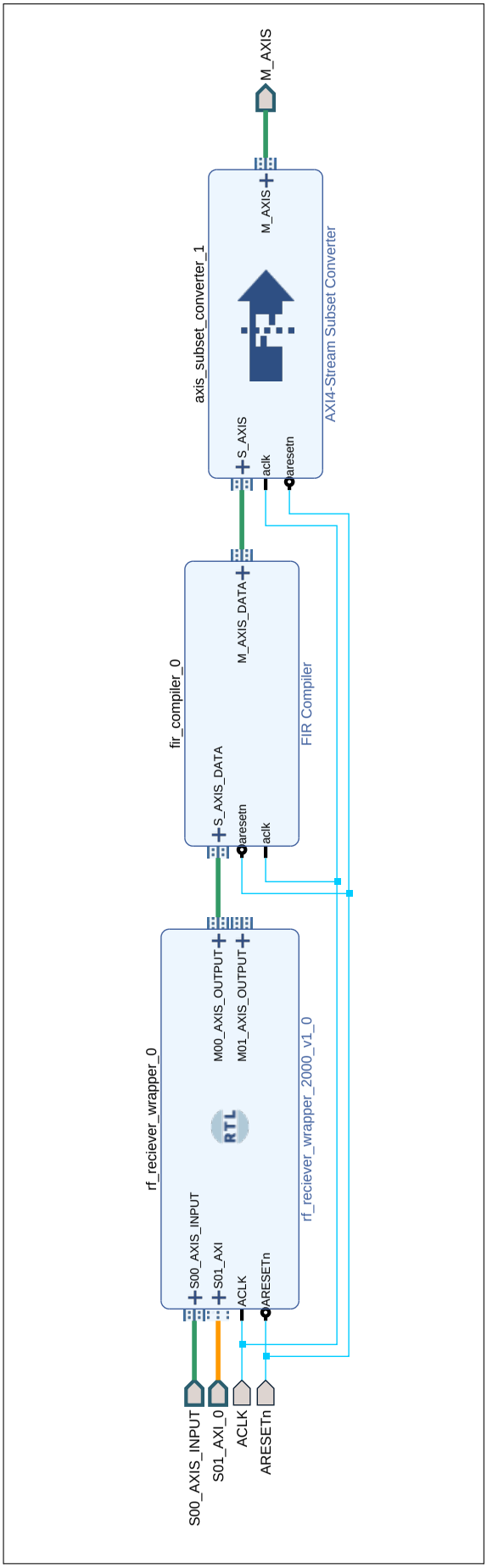

Abbildung 5.2: Block-Design des Sub-Blocks: RF Channel 0

Blockdesign: ADC

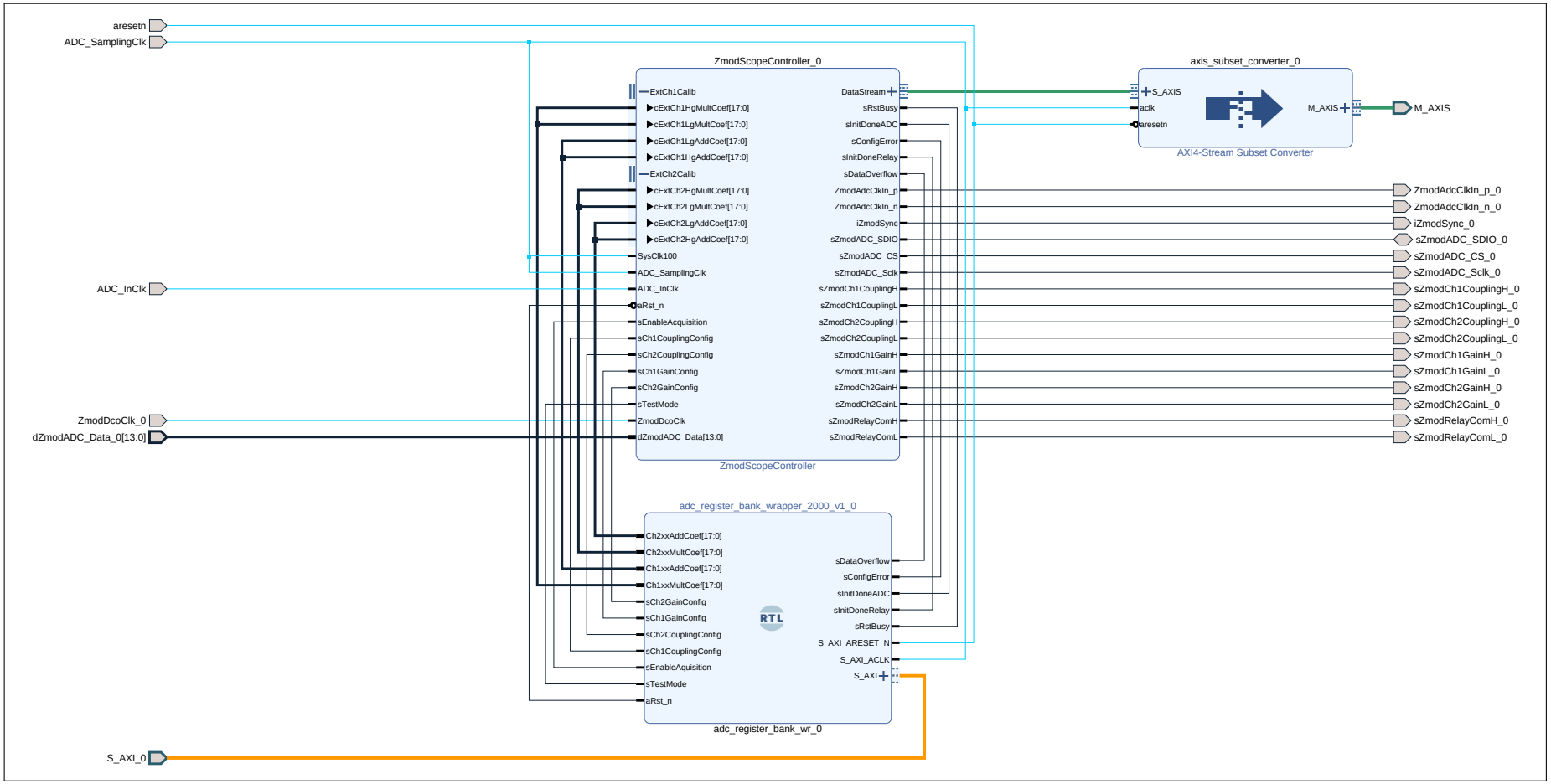

Abbildung 5.3: Block-Design des Sub-Blocks: ADC

5.2 Simulationsergebnis des RF-Recievers

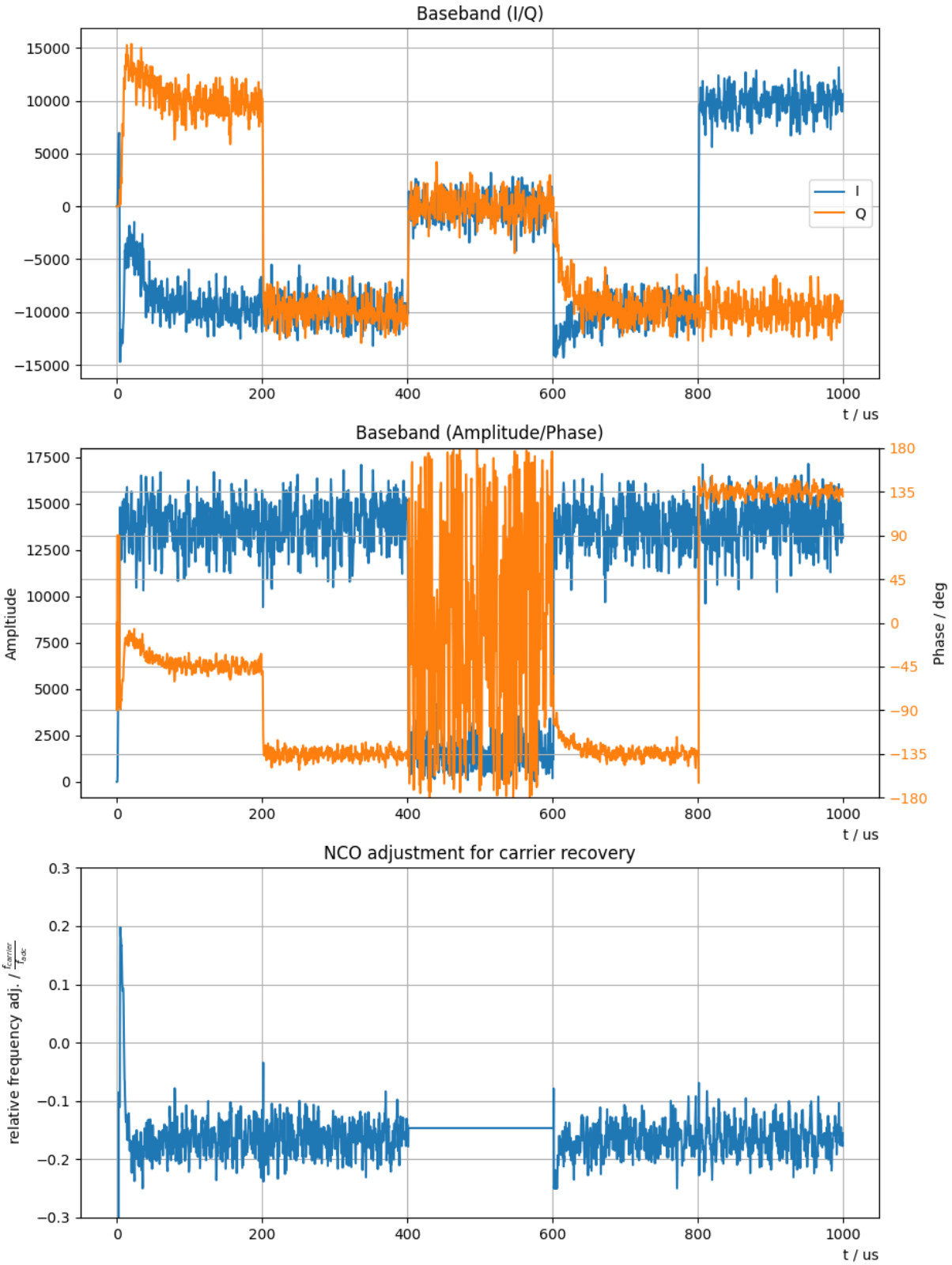

Abbildung 5.4: Ergebnis der Simulation des RF-Recievers für: $f_c = 20\,\mathrm{MHz}, \mathrm{SNR} = 3\,\mathrm{dB}, A = -6\,\mathrm{dBFS}, \frac{\Delta f}{f_c} = -800\,\mathrm{ppm}$

Abkürzungsverzeichnis

FPGA Field-programmable gate array
SDR Software defined radio
ADC Analog-to-digital converter
BPSK Binary Phase-Shift Keying
QPSK Quadratur Phase-Shift Keying
CPU Central Processing Unit
NCO Numerically Controlled Oscillator
AXI Advanced eXtensible Interface
DMA Direct Memory Access
APSoC All programmable System on a Chip
BRAM Block random access memory
CLB Configurable logic block
DSP Digital Signal Processing
IOB Input/Output-Buffer
DAC Digital-to-analog converter
CIC Cascaded-Integrator-Comb
FIR Finite impulse response
PID Propertional-Integral-Differential
NCO Numerically controlled oscillator
DDS Direct digital synthesis
PYNQ Python for Zynq
VHDL Very High Speed Integrated Circuit Hardware Description Language
RTL Register transfer level
ZFS Zettabyte File System
FSM Finite state machine
FSK Frequency shift keying

Abbildungsverzeichnis

Tabellenverzeichnis