



**BERGISCHE  
UNIVERSITÄT  
WUPPERTAL**

**Fachbereich E**

Elektrotechnik, Informationstechnik, Medientechnik

---

Lehrstuhl für Nachrichtentechnik / Audiosignalverarbeitung und InCar Noise Control

# Bachelor-Thesis

**Implementierung einer FPGA-basierten ADAT-Schnittstelle  
für DSP der C6000-Reihe**

Simon Lohmann

1023814

Informationstechnologie

Systems & Components

Wuppertal, den 28. Januar 2014

Betreuer: Prof. Dr.Ing. Detlef Krahé

Erstgutachter: Prof. Dr.Ing. Detlef Krahé

Zweitgutachter: Prof. Dr.Ing. Carsten Gremzow



BACHELOR-THESIS

Kandidat: **Lohmann, Simon**  
Matrikelnummer: **1023814**  
Studienabschluss: **Bachelor Informationstechnologie**  
Betreuer: **Prof. Dr.-Ing. Detlef Krahé**

Thema: **Implementierung einer FPGA-basierten ADAT-Schnittstelle für DSP der C6000-Reihe**

Erläuterungen:

In der akustischen Forschung werden multiple-input multiple-output (MIMO) Systeme oftmals auf Basis moderner DSP-Architekturen (Digital Signal Processor) erstellt. Verschiedene Hersteller bieten DSP-Entwicklungs-Lösungen in Form von Platinen an, die einen Großteil der notwendigen Peripherie zur Verfügung stellen. Häufig sind auf den Boards für Audio-Anwendungen auch Codecs enthalten, in der Regel in zweikanaliger Ausführung. Für mehrkanalige Audio-Anwendungen stehen aus dem Bereich der professionellen Audiotechnik Wandlersysteme zur Verfügung, die zwar nicht direkt an ein derartiges DSP-Board angeschlossen werden können, welche aber über eine ADAT-Schnittstelle (Alesis Digital Audio Tape) verfügen. ADAT hat sich als ein Standard etabliert und wird von einer Vielzahl der Wandlersysteme unterstützt.

In der Arbeit soll mit Hilfe eines FPGA (Field Programmable Gate Array) die Anbindung eines mehrkanaligen Audio-Interface an einen Texas Instruments TMS320C6455 DSP über die ADAT-Schnittstelle ermöglicht werden. Der Entwurf soll die Ausweitung auf bis zu drei ADAT-Kanäle berücksichtigen.

Wuppertal, den 28. Oktober 2013

  
(Unterschrift)

Erstgutachter: **Prof. Dr.-Ing. D. Krahé**  
Zweitgutachter: **Prof. Dr.-Ing. C. Gremzow**

Prüfungsamt:

Kennziffer: **BTHB JT 6 JK**  
Ausgabedatum: **28.10.13**  
Abgabedatum: **28.01.14**

  
(Unterschrift)

## Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Wuppertal, den 28. Januar 2014

---

(Unterschrift)

## Einverständniserklärung

Ich bin damit einverstanden, dass meine Abschlussarbeit wissenschaftlich interessierten Personen oder Institutionen zur Verfügung gestellt werden kann. Korrektur- oder Bewertungshinweise in meiner Arbeit dürfen nicht zitiert werden.

Wuppertal, den 28. Januar 2014

---

(Unterschrift)

## Kurzfassung

Bei der Erforschung von Active-Noise-Control (ANC)-Systemen wird eine hohe Zahl von Ein- und Ausgängen für die zahlreichen beteiligten Audiosignale benötigt. Die Berechnungen des ANC-Systems erfolgen in der Regel auf digitalen Signalprozessoren (DSP).

Im Rahmen dieser Thesis wurde eine Schnittstelle zwischen dem in der professionellen Audiotechnik etablierten ADAT-Format, welches Audiosignale in digitaler Form über Lichtleiter überträgt, und der McBSP-Schnittstelle der DSPs aus der C6000-Reihe von TEXAS INSTRUMENTS entwickelt. Die Umwandlung der Formate erfolgt in einem Field-Programmable-Gate-Array (FPGA).

Das resultierende VHDL-Design ist problemlos auf eine beliebige Zahl von ADAT-Schnittstellen skalierbar<sup>1</sup>, kann sowohl als ADAT-Master als auch als ADAT-Slave arbeiten und passt sich automatisch sowohl an beliebige ADAT-Abtastraten als auch an beliebige Übertragungsraten der McBSP-Schnittstelle an.

Die Kombination aus ADAT-De- und Enkoder inklusive dazwischengeschaltetem Puffer erreicht eine Latenz von zwei ADAT-Frames (bei einer Abtastrate von 48 kHz entspräche das nur ca. 0,042 ms). Bei der Verbindung zum McBSP hängt die Latenz vor allem von der gewählten Verbindungsgeschwindigkeit ab, beträgt aber ebenfalls nie mehr als 2 ADAT-Frames.

Zum Schluss der Thesis wurde das gesamte System mit jeweils 3 ADAT Ein-/Ausgängen als Aufsteckplatine für die DSP-Entwicklungsboards TMS320C6455 DSK und TMS320C6713 DSK von SPECTRUM DIGITAL entwickelt und gebaut.

---

<sup>1</sup>Wobei natürlich der FPGA ebenfalls entsprechend skaliert werden muss.

---

## Abstract

In the research on Active Noise Control (ANC)-systems, a high number of in- and outputs are necessary for the audiosignals. The ANC-calculations are usually made on digital signal processors (DSP).

In this thesis, an interface between the ADAT-format widely used in professional audio applications (which transmits digital audiosignals over fiber optics) and the McBSP-interface from the DSPs of TEXAS INSTRUMENTS C6000 series was developed. The format conversions are made using a field-programmable-gate-array (FPGA).

The resulting VHDL-design is easily scalable to an arbitrary number of ADAT-interfaces<sup>2</sup>, can work as ADAT-master or ADAT-slave and automatically matches itself to any used ADAT-samplerate as well as any McBSP-datarate.

The combination of ADAT-decoder and encoder including the buffer in between both reaches a latency of only two ADAT-frames (at a samplerate of 48 kHz, this would result in approximately 0,042 ms). In the connection to the McBSP, the latency is mainly dependent on the chosen datarate. However it can never be greater than two ADAT-frames.

Finally, a daughtercard for the DSP-developmentboards TMS320C6455 DSK and TMS3206713 DSK by SPECTRUM DIGITAL with three ADAT-interfaces was developed and built.

---

<sup>2</sup>Of course, the FPGA has to be scaled accordingly.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problemstellung & Ziele . . . . .	1
1.3 Aufbau der Thesis . . . . .	2
<b>2 ADAT</b>	<b>3</b>
2.1 Geschichte . . . . .	3
2.1.1 ADAT - Der Mehrspurrekorder . . . . .	3
2.1.2 ADAT-Sync . . . . .	4
2.1.3 ADAT-Lightpipe . . . . .	5
2.2 Aufbau des ADAT-Lightpipe-Protokolls . . . . .	5
2.2.1 Frameaufbau . . . . .	5
2.2.2 User Bits . . . . .	5
2.2.3 Audio-Daten . . . . .	6
2.2.4 Bitkodierung, Takt & Frame-Synchronisation . . . . .	7
2.3 Taktgewinnung . . . . .	7
2.3.1 Methode aus dem Patent US 5,297,181 . . . . .	7
2.3.2 Taktgewinnung in einem reinen Digitalsystem . . . . .	11
<b>3 McBSP - Die Verbindung mit dem DSP</b>	<b>15</b>
3.1 Hardware . . . . .	15
3.2 Frames, Phasen und Elemente . . . . .	15
3.3 Ansteuerung im DSP . . . . .	16
3.3.1 Ansteuerung: Manuell . . . . .	16
3.3.2 Ansteuerung: CSL . . . . .	16
3.4 Ansteuerung im FPGA . . . . .	17
3.4.1 Warum auch die Userbits? . . . . .	18
3.4.2 Auffüllen auf 24 Bit pro Element . . . . .	18
3.4.3 Programmablauf im DSP . . . . .	19
3.4.4 Senden . . . . .	20
3.4.5 Empfangen . . . . .	21

---

<b>4 Realisierung in einem FPGA</b>	<b>24</b>
4.1 Wahl des konkreten FPGAs . . . . .	24
4.2 Allgemeines zu VHDL . . . . .	25
4.2.1 Entity . . . . .	25
4.2.2 Architecture . . . . .	26
4.2.3 Signale, Variablen und Konstanten . . . . .	27
4.2.4 Prozesse . . . . .	29
4.2.5 Komponenten . . . . .	30
4.2.6 Generate . . . . .	31
4.2.7 Packages . . . . .	32
4.2.8 Testbenches . . . . .	32
4.2.9 Das Programm „Tektronix CSV -> VHDL Testbench“- Verwenden von real aufgezeichneten Daten für die Simulation . . . . .	33
4.3 Constraints . . . . .	36
4.3.1 Die .ucf-Datei . . . . .	36
4.3.2 Constraints im VHDL-Code . . . . .	39
4.4 Synthese . . . . .	40
4.4.1 Nicht synthetisierbare Konstrukte . . . . .	40
4.4.2 VHDL-Konstrukte, die bei der Synthese verloren gehen . . . . .	41
4.4.3 Synchron & Asynchron - Die Gefahr metastabiler Zustände . . . . .	41
4.4.4 Ressourcen-Optimierung . . . . .	42
4.5 Map und PAR . . . . .	43
4.5.1 Map . . . . .	43
4.5.2 PAR . . . . .	43
4.5.3 Ressourcen-Optimierung . . . . .	44
4.6 Die VHDL-Module . . . . .	44
4.6.1 Übersicht . . . . .	44
4.6.2 Das Package ADAT . . . . .	45
4.6.3 ADAT_De_Enkoder . . . . .	47
4.6.4 ADAT_Taktgewinnung . . . . .	48
4.6.5 ADAT_Dekoder . . . . .	49
4.6.6 ADAT_Enkoder . . . . .	50
4.6.7 ADAT_BLOCKRAM . . . . .	52
4.6.8 DynamischerADATPuffer_BRAM . . . . .	53
4.6.9 McBSP_Interface . . . . .	56
<b>5 Realisierung der Hardware</b>	<b>57</b>
5.1 Die Schaltung . . . . .	57
5.1.1 Der FPGA . . . . .	57
5.1.2 ADAT-Schnittstellen . . . . .	60
5.1.3 Wahl der ADAT-Taktquelle . . . . .	62

---

5.1.4	Status-LEDs . . . . .	65
5.2	Die Platine . . . . .	67
5.2.1	HF-Betrachtung nötig? . . . . .	67
5.2.2	Der FPGA . . . . .	69
5.2.3	Datenleitungen . . . . .	70
5.2.4	Separate Analog-Flächen für die ADAT-Buchsen . . . . .	71
5.2.5	Kühlung des Spannungsreglers . . . . .	71
5.2.6	Heraus geführte Signale . . . . .	74
5.2.7	Testpunkte . . . . .	74
<b>6</b>	<b>Analyse</b>	<b>76</b>
6.1	ADAT-Loopback . . . . .	76
6.2	Kommunikation über den McBSP . . . . .	77
6.3	Taktgewinnung . . . . .	77
<b>7</b>	<b>Schlussbetrachtung</b>	<b>78</b>
7.1	Fazit . . . . .	78
7.2	Ausblick . . . . .	78
<b>A</b>	<b>Literatur</b>	<b>79</b>
<b>B</b>	<b>Abbildungsverzeichnis</b>	<b>83</b>
<b>C</b>	<b>Tabellenverzeichnis</b>	<b>84</b>
<b>D</b>	<b>Listings</b>	<b>85</b>
<b>E</b>	<b>Anhang</b>	<b>86</b>
E.1	Schaltplan . . . . .	86
E.2	Platinenlayout . . . . .	92
E.3	Heraus geführte FPGA-Pins . . . . .	93
E.4	Testpunkte . . . . .	94
E.5	Taktrouting im FPGA . . . . .	95

# 1 Einleitung

## 1.1 Motivation

In ANC (Active Noise Control)-Systemen wird ein vorhandenes (störendes) Schallfeld mit einem Array von Mikrofonen aufgenommen. Durch das Senden passend berechneter Gegensignale über ein Lautsprecher-Array lässt sich nach dem Prinzip der destruktiven Interferenz das ursprünglich vorhandene Schallfeld dämpfen.

Dabei ist eine geringe Latenz zwischen der Aufnahme des Störsignals und dem Senden des Gegensignals für die Funktion des ANC-Systems essentiell.

Die Berechnung des Gegensignals erfolgt aufgrund der erforderlichen Rechenleistung in der Regel auf digitalen Signalprozessoren (DSP).

Als Problemstellung ergibt sich nun der Anschluss einer großen Menge von Mikrofonen und Lautsprechern (bzw. entsprechender Audio Ein- und Ausgänge) an den DSP – bei gleichzeitig hoher Audioqualität und geringer Latenz.

In diesem Fall werden Wandler aus dem Studiobereich verwendet, welche bereits die entsprechend hochwertigen Wandler enthalten und über die optische ADAT-Schnittstelle verfügen, welche bis zu 8 Audiokanäle über einen Lichtwellenleiter überträgt.

Das Audiosignal kann allerdings nicht ohne weiteres direkt an den DSP angebunden werden, da zunächst Empfang und Dekodierung des ADAT-Signals erforderlich sind. Die einzige bekannten ADAT De- bzw. Enkoder ICs AL1401AG und AL1402G von WAVEFRONT sind inzwischen allerdings abgekündigt: „This part is End Of Life. Purchase is no longer available for AL1401AG. Last time purchase date for AL1402G is November 30, 2013.“ [Wav].

## 1.2 Problemstellung & Ziele

Ziel dieser Thesis ist es nun, eine ADAT-Schnittstelle für die in diesem Fall verwendeten DSPs der C6000-Reihe von TEXAS INSTRUMENTS zu entwickeln, wobei die Implementierung der Schnittstelle in einem FPGA (Field Programmable Gate Array) erfolgt.

Als konkretes Problem beim ADAT-Empfang ergibt sich einerseits das Extrahieren von Takt und Frame-Sync aus dem empfangenen Datenstrom und andererseits die Dekodierung des eigentlichen Datenstroms.

---

Bei der Kommunikation mit dem DSP ist wiederum ein Zusammenführen/Zerlegen aller Daten und eine schnelle Übertragung dieser vom FPGA zum DSP (und wieder zurück) nötig.

Berücksichtigt werden soll dabei ein Ausbau auf bis zu drei ADAT-Schnittstellen, ein entsprechend leicht anpassbares Design ist also erstrebenswert.

## 1.3 Aufbau der Thesis

Der Hauptteil dieser Thesis ist in 4 Kapitel unterteilt, die die einzelnen Schritte bei der Realisierung des Projektes darstellen. Am Anfang des Kapitels werden dabei stets die Grundlagen der jeweiligen Thematik erläutert.

In Kapitel 2 „ADAT“ wird zunächst die historische Entwicklung von ADAT beschrieben, danach folgen die Grundlagen des Protokolls. Zum Schluss wird neben der „Originalmethode“ noch eine selbst entwickelte Variante zur Taktgewinnung vorgestellt.

Kapitel 3 „McBSP - Die Verbindung mit dem DSP“ beschäftigt sich mit der McBSP-Einheit der DSPs der C6000-Reihe. Hierbei wird insbesondere auf die Ansteuerung im FPGA eingegangen.

Kapitel 4 „Realisierung in einem FPGA“ erläutert zunächst die diversen Grundlagen und Abläufe bei der FPGA-Entwicklung. Zum Schluss werden die entwickelten VHDL-Module vorgestellt.

In Kapitel 5 „Realisierung der Hardware“ wird die Realisierung der Schnittstelle auf einer Platine behandelt.

In Kapitel 6 „Analyse“ werden die letztendlich erreichten Latenzwerte der Hardware analysiert. Zum Schluss wird in Kapitel 7 „Schlussbetrachtung“ eine kurze Zusammenfassung der erreichten bzw. nicht erreichten Ziele gegeben, gefolgt von Verbesserungsvorschlägen für eine neue Platinenrevision.

## 2 ADAT

Dieses Kapitel ist in drei Teile unterteilt:

- In Abschnitt 2.1 „*Geschichte*“ wird erläutert was mit dem Namen „ADAT“ ursprünglich gemeint war und wie sich die Bedeutung gewandelt hat. Außerdem wird auf die Entwicklung der technischen Details von ADAT eingegangen.
- Abschnitt 2.2 „*Aufbau des ADAT-Lightpipe-Protokolls*“ geht auf das Protokoll von ADAT ein und erklärt dessen Takt- und Synchronisationsmechanismus.
- In Abschnitt 2.3 „*Taktgewinnung*“ werden zwei Methoden zur Taktgewinnung aus dem ADAT-Signal vorgestellt: Einerseits die analoge Methode aus dem originalen „ADAT-Patent“ US 5,297,181 [Bar+94], andererseits eine selbst entwickelte Variante zur Verwendung in rein digitalen Systemen.

### 2.1 Geschichte

#### 2.1.1 ADAT - Der Mehrspurrekorder

Ursprünglich war ADAT (Alesis Digital Audio Tape) der Name eines digitalen Mehrspurrekorders der Firma ALESIS, der 1991 vorgestellt wurde. Dieser zeichnete 8 Audiospuren mit einer regulären Abtastrate von 48 kHz (diese war auf Wunsch variierbar) bei einer Auflösung von 16 Bit pro Sample auf. [Ale, S. 9]

Aufzeichnungsmedium war die damals verhältnismäßig kostengünstige S-VHS-Kassette, welche 2001 mit dem ADAT HD24 von der Festplatte abgelöst wurde. Inzwischen hat sich im Studio die DAW (Digital Audio Workstation)<sup>1</sup> durchgesetzt, welche wesentlich flexibler einsetzbar ist und nicht nur die Aufzeichnung, sondern zusätzlich auch die Nachbearbeitung der Audiosignale ermöglicht.

**Entwicklung von Sample-Auflösung und Abtastrate** Es gibt inzwischen drei ADAT-Klassen (ADAT Typ I, ADAT Typ II und eine dritte Variante, die keinen Namen trägt), welche sich jeweils in der Auflösung der Audiosamples unterscheiden.

<sup>1</sup>DAW: Programm, welches sämtliche Verarbeitungsschritte von der Aufzeichnung über das Arrangieren bis hin zu Nachbearbeitung, Mischen und Mastern von Audiomaterial ermöglicht.

---

Bei der Definition der beiden Klassen gehen die Informationen allerdings auseinander:

Laut dem Datenblatt des (einzigsten) ADAT-Enkoders AL1401AG [Wav05a, S. 5] werden bei ADAT Typ I für die Audiosignale 16 Bits gesendet. Später wurde demnach die Auflösung auf 20 Bit pro Sample erhöht (ADAT Typ II) [Wav05a, S. 5]. Nach der „ADAT“-Appnote von NTI AUDIO hingegen überträgt Typ I bereits 20 Bit, während Typ II 24 Bit nutzt. [NTi12, S. 3]

Laut der englischen Bedienungsanleitung des ADAT HD24 von ALESIS, also der vermutlich zuverlässigsten Quelle, werden Typ I und Typ II wie im Datenblatt des oben genannten ADAT-Enkoders definiert.

Interessanterweise werden seit jeher 24 Bit übertragen, und die nicht genutzten Bits dabei einfach mit „0“ aufgefüllt (Senden) bzw. verworfen (Empfang) [NTi12, S. 2][Ale01a, S. 25]. Heute sind Interfaces mit 24 Bit pro Sample üblich.

Mit der Zeit verbesserte sich auch die nutzbare Abtastrate. So konnte das ADAT HD24 (als letzter ADAT-Rekorder von Alesis) wahlweise 24 Kanäle bei 48 KHz oder 12 Kanäle bei 96 KHz, jeweils mit 24 Bits pro Sample aufzeichnen.

### 2.1.2 ADAT-Sync

ADAT-Sync ist eine zusätzliche Synchronisationsmöglichkeit, herausgeführt als neunpolige D-SUB Schnittstelle. Die Funktionsweise wird in Patent EP 0 621 976 B1[Bar+99] beschrieben.

→ Laut [Ada] ist ist die dort beschriebene Pinbelegung allerdings nicht identisch mit der in realen ADAT-Geräten tatsächlich verwendeten Pinbelegung.

Wesentliche Funktion von ADAT-Sync ist der synchrone Betrieb von ADAT-Audio-Rekordern, d.h. die Synchronisierung der Steuerbefehle und der Aufnahme. Dazu wird unter anderem ein absoluter Timecode übertragen.

Eine typische Anwendung wäre die Verwendung von z.B. drei ADAT-Rekordern parallel, die durch das erste Gerät gesteuert werden. Die Sync-Schnittstellen der ADAT-Rekorder werden seriell verschaltet, dabei sind je nach Rekorder bis zu 16 parallele Geräte möglich<sup>2</sup>

---

<sup>2</sup>Beim ersten ADAT-Rekorder (8 Spuren pro Gerät) waren es bis zu 16 Geräte, also 128 Audio-Spuren [Ale, S. 38]. Beim HD24 sind es bis zu fünf Geräte, wodurch sich laut englischer Bedienungsanleitung 120 Spuren ergeben [Ale01a, S. 24]. In der deutschen Bedienungsanleitung ist zwar von 1220 Spuren die Rede[Ale01c, S. 25], da aber ebenfalls nur fünf Geräte erwähnt werden, ergeben sich auch hier die  $120 = 5 \cdot 24$  Kanäle aus der englischen Anleitung. – Auch sonst scheint die deutsche Anleitung fehlerbehaftet zu sein: So ist sie laut Seite 2 eine Übersetzung des angeblich erst ein Jahr später (2002) erschienenen englischen Originals, welches laut Original aber im selben Jahr (2001) erschien.

### 2.1.3 ADAT-Lightpipe

*Wird heute in der Regel nur noch als „ADAT“ bezeichnet*

Seit dem ersten ADAT-Gerät gibt es die Möglichkeit, die Audio-Daten über einen Lichtleiter zu übertragen, damals noch um Audio-Spuren zwischen den Geräten überspielen zu können. Heute haben auch zahlreiche Mischpulte und Audio-Wandlerkarten eine ADAT-Lightpipe-Schnittstelle.

Das Format überträgt pro Schnittstelle 8 Kanäle mit 24 Bit pro Audiosample, wobei Geräte mit geringerer Auflösung die fehlenden Bits der Samples einfach mit Nullen auffüllen [Ale01a, S. 25]. Die Abtastrate ist üblicherweise 44,1 oder 48 KHz, die Schnittstelle ermöglicht aber auch Abweichungen davon:

Der ADAT-Dekoder-Chip AL1402G und der dazugehörige Enkoder AL1401AG von WAVEFRONT erlauben beispielsweise einen Bereich von 30 bis 55 kHz [Wav05b, S. 3][Wav05a, S. 3]. Auch beim ADAT-Mehrspur-Rekorder (s.o.) konnte die Abtastrate schon eingestellt werden: In der Anleitung des Mehrspurrekorders ADAT finden sich allerdings widersprüchliche Angaben zum konkreten Frequenzbereich: Auf Seite 9 wird von 42,7 bis 50,85 kHz gesprochen, auf Seite 54 sind es 40,4 bis 50,8 kHz. [Ale]

Später wurde außerdem die Möglichkeit hinzugefügt, zwei der Kanäle als einen Kanal zu interpretieren und damit die Abtastrate zu verdoppeln (siehe 2.2.3.1 *S/Mux: Doppelte Abtastrate*).

Hardwaremäßig entspricht ADAT-Lightpipe der S/PDIF (Sony/Philips Digital Interface Format)-Schnittstelle, es wird allerdings ein anderes Protokoll verwendet.

## 2.2 Aufbau des ADAT-Lightpipe-Protokolls

ADAT Lightpipe ist eine Frame-basiertes Protokoll. Für jedes Audio-Sample wird dabei ein ADAT-Frame gesendet, der die Audiodaten mehrerer Kanäle vereint:

### 2.2.1 Frameaufbau

Ein ADAT-Frame besteht aus 256 Bits, wobei nur 196 davon Daten sind. Der Frameaufbau ist in Abbildung 2.1 zu sehen, eine Übersicht über die verschiedenen Bit-Typen in Tabelle 2.1.

### 2.2.2 User Bits

Über die User-Bits können zusätzlich zum Audiosignal weitere Informationen übertragen werden (bei 48 kHz Abtastrate z.B. pro Userbit mit 48.000 Bit/s). In Tabelle

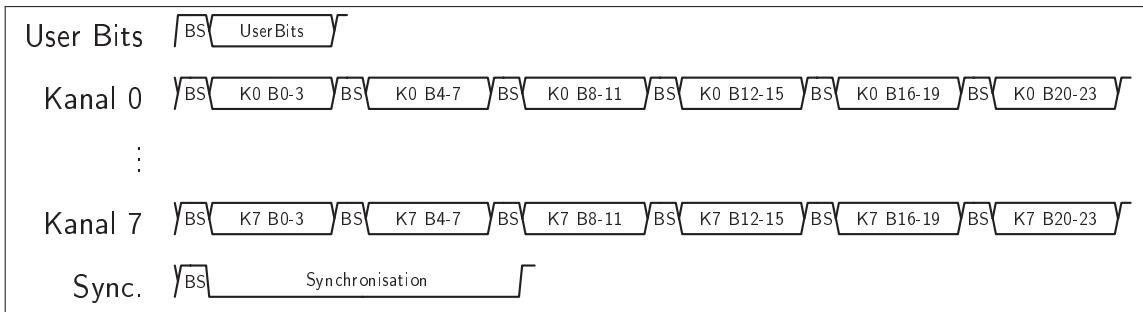


Abbildung 2.1: Frameaufbau ADAT (BS: Bitstuffing, K0 B0-3: Audiodaten Kanal 0 - Bits 0 bis 3)

	Bits	Name	Beschreibung
<b>Daten</b>	4	User Bits	Enthält Informationen zu den Audiodaten. Kann zusätzlich auch MIDI übertragen.
	192	Audio	8 Kanäle mit je 24 Bit, Signed, MSB-First
<b>Sync. &amp; Takt</b>	50	Bitstuffing	'1' nach 4 Datenbits (außer im Sync.-Signal)
	10	Synchronisation	10 mal '0'

Tabelle 2.1: Bittypen im ADAT-Frame

2.2 sind die einzelnen Funktionen der Userbits aufgeführt.

Bit Nr.	Bedeutung	Beschreibung
0	Timecode	32-Bit-Timecode [Wav05a, S. 5][Ada]
1	MIDI	MIDI-Daten
2	S/Mux	„0“ :Normal, „1“ :S/Mux, d.h. doppelte Abtastrate aber nur noch 4 Kanäle
3	Reserviert	„0“

Tabelle 2.2: Die ADAT-Userbits

### 2.2.3 Audio-Daten

Die Audiodaten werden unabhängig von der tatsächlichen Auflösung/Samplerate der Geräte immer als 8 Kanäle mit je 24 Bit pro Sample übertragen, welche dann entsprechend der Fähigkeiten des Gerätes interpretiert werden.

Unterstützt das Gerät nur 16 bzw. 20 Bit (ADAT Typ I & II), so werden die niedrigerwertigen Bits beim Senden mit Nullen aufgefüllt. Beim Empfang werden eventuell

---

nicht nutzbare, aber empfangene Bits einfach abgeschnitten. [Ale01a, S. 25]

### 2.2.3.1 S/Mux: Doppelte Abtastrate

Mit der Erweiterung der (nicht öffentlich zugänglichen) ADAT-Spezifikation im Jahre 2001 durch [Ale01b] (öffentlicht zugänglich) wurde das „S/Mux“-Bit eingeführt – damals lediglich als Userbit „U2“ bezeichnet. Inzwischen hat sich die Bezeichnung „S/Mux“ etabliert [Wav05a, S. 2].

Ist in den Userbits S/Muxaktiviert, so werden demnach die Samples von zwei Kanälen als aufeinander folgende Samples eines Kanals interpretiert, wodurch sich die effektive Abtastrate der jetzt nur noch vier Kanäle verdoppelt.

Dieses Vorgehen hat nach [Ale01b, S. 2] außerdem den Vorteil, dass mit Geräten, die kein S/Muxunterstützen, das Signal dennoch problemlos empfangen werden kann, da jeder Kanal normal interpretiert wird – also so als wäre jedes zweite Sample weggelassen worden – effektiv also ein sehr rudimentäres Downsampling auf die halbe Abtastrate stattfindet.

### 2.2.4 Bitkodierung, Takt & Frame-Synchronisation

Die Übertragung bei ADAT erfolgt NRZI-M (Non-Return-to-Zero-Inverted-Mark)-kodiert, d.h. bei einer „1“ im Ursprungssignal ändert sich der Pegel des kodierten Signals, bei einer „0“ bleibt er gleich.

Durch die Übertragung über einen einzelnen Lichtleiter ist kein separates Taktsignal verfügbar. Stattdessen wird das Taktsignal in das Datensignal integriert. Um genug Signalwechsel für die Taktgewinnung zu erzeugen, wird nach jeweils 4 Datenbits eine „1“ eingefügt (Bitstuffing).

Für die Frame-Synchronisierung wird dieses Muster mit einer Folge von 10 Nullen am Ende des Frames bewusst unterbrochen.

## 2.3 Taktgewinnung

Im Folgenden werden zwei Methoden zur Taktgewinnung aus einem ADAT-Signal beschrieben: Erstens die „Original“-Variante, die teilweise auf Analogtechnik basiert und zweitens eine selbst entwickelte Methode zur Verwendung in reinen Digitalsystemen.

### 2.3.1 Methode aus dem Patent US 5,297,181

Im Patent US 5,297,181[Bar+94] – dem ursprünglichen „ADAT-Patent“ – wird unter anderem Methode zur Taktgewinnung aus dem ADAT-Signal beschrieben, welche

---

im Wesentlichen darauf basiert, dass ein VCO (Voltage Controlled Oscillator) in mehreren Stufen an den korrekten Takt angepasst wird.

*Bei den folgenden Abschnitten sollte im Hinterkopf behalten werden, dass die Einheiten Synchronisierungssignal-Erkennung und Taktgenerierung bei dieser Methode nicht wirklich getrennt gesehen werden können, da sie auf einander basieren und sich gegenseitig beeinflussen.*

### 2.3.1.1 Erkennung des Synchronisierungssignals

Die in [Bar+94] vorgestellte Methode für einen ADAT-Takt-Empfänger kann in der Anfangsphase der Taktgewinnung noch keine phasenrichtige Erkennung des Synchronisierungssignals sicherstellen, da der VCO-Takt noch nicht korrekt eingestellt wurde. Nachdem alle Stufen der Taktgewinnung durchlaufen wurden und Frequenz sowie Phase von VCO und Eingangssignal zusammenpassen, ließe sich an Zähler 613 in Figur 6(A) ein passendes Sync.-Signal abgreifen, wenn dieser Überläuft.

Auch der in Figur 7 (hier Abbildung 2.3) von [Bar+94] aufgeführte „Frame-Detector“ (702) wäre vom Namen her ein guter Kandidat für eine Einheit zur Erkennung des Sync-Signals – an diesem kann man schließlich den Frame-Anfang erkennen – scheint allerdings eine andere Funktionalität zu haben, als der Name vermuten ließe:

The digital data signal 712 is connected to the input of pulse generator 701. Pulse generator 701 receives the NRZI digital data signal from the interface transmission system and outputs a pulse every time it detects a transition in signal 712. The output of pulse generator 701 is coupled to frame detector 702, data separator 703 and phase detector 704. Frame detector 702 monitors the number of transitions in the data signal 712, and outputs a signal after every 256 transitions. [Bar+94, Spalte 14, Zeile 58 ff.]

Mit anderen Worten: Der „Frame-Detector“ gibt jedes mal ein Signal aus, wenn er 256 Transitionen im Eingangssignal gezählt hat. Dies hat allerdings keinen Zusammenhang mit dem Anfang eines Frames – dieser hat zwar 256 Bits, aber nie 256 Transitionen – der Name „Frame-Detector“ ist hier also falsch gewählt.

Der „Frame-Detector“-Ausgang ist mit dem „Priority selector“ verbunden. Dieser bestimmt, ob sich das System in der Synchronisierungsstufe 1 oder 2 befindet. Außerdem gibt er ein Signal zur Anpassung der VCO-Frequenz aus. Eine Einheit mit identischer Funktionsweise und Verwendung des Ausgangssignals wie beim „Frame-Detector“ wird in Spalte 13 beschrieben:

An 8 bit transition counter is incremented at the receipt of each channel transition [...] .

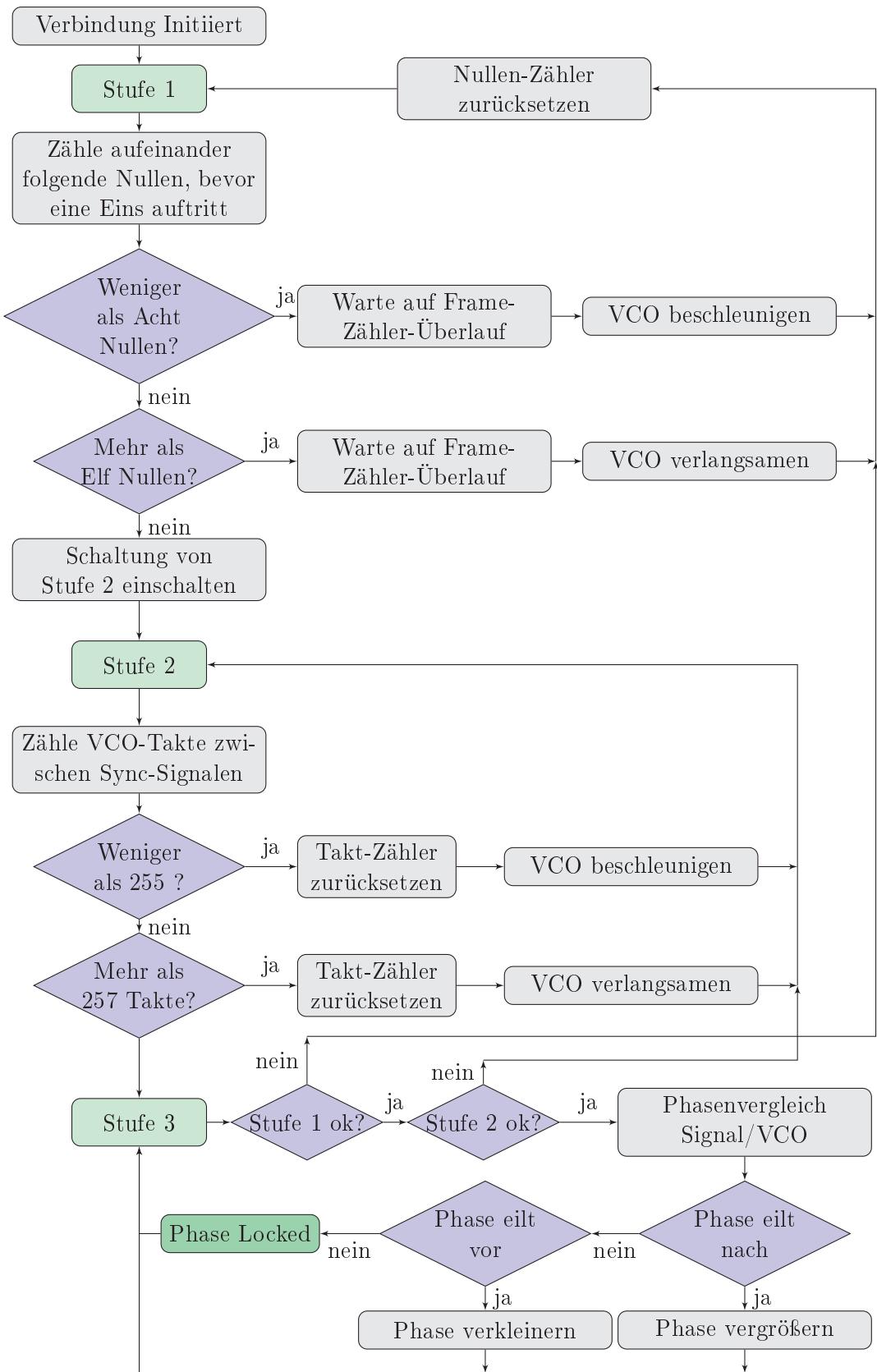


Abbildung 2.2: Taktgewinnung gemäß Patent US 5,297,181, Fig 6A &amp; 6B

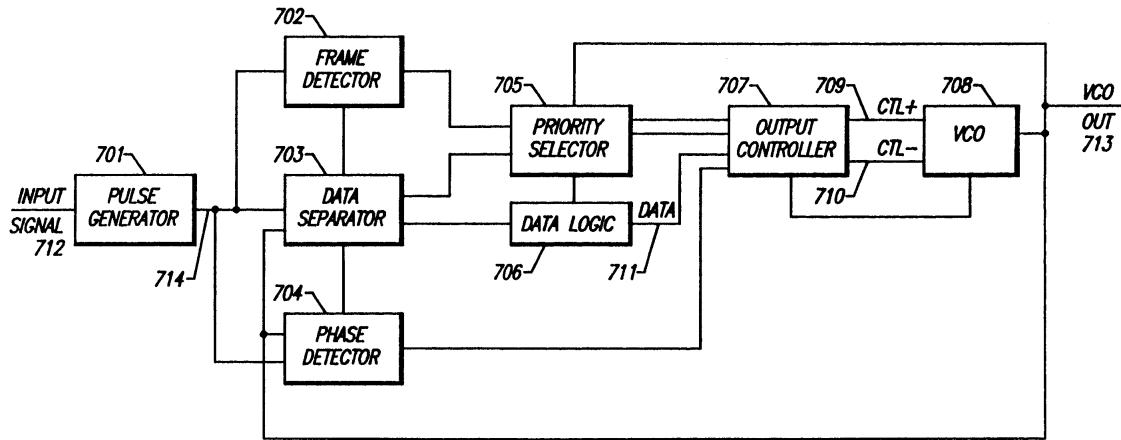


Abbildung 2.3: Figur 7 aus dem ADAT-Patent [Bar+94]

Another counter is arranged to count the number of continuously received zeros (driven by the receive VCO), and the maximum count is made to set flip-flops, depending on established limits (i.e., maximum of 12, minimum of 8 continuous zeros). The values of these flip flops are clocked into registers at the rollover of the transition counter to indicate the degree and direction of any gross VCO error. If the continuous zero count got to eight, but never got to 13, then the VCO is within stage 1 limits, and control is passed to stage 2. [Bar+94, Spalte 13, Zeile 26 ff.]

Es handelt sich bei dem „Frame-Detector“ in Abbildung 2.3 also offensichtlich nicht um eine Einheit zur Erkennung des Frame-Anfangs, sondern vielmehr um einen einfachen 8-Bit-Transitionszähler. Dieser gibt der nachfolgenden Einheit lediglich einen Hinweis, dass sie die gemessene Synchronisierungssequenzlänge jetzt in ein Register übernehmen kann<sup>3</sup>.

Über die tatsächlich verwendete Frame-Erkennungsmethode sind keine Informationen in [Bar+94] vorhanden.

### 2.3.1.2 Taktgewinnung

Der Takt wird in drei Stufen aus dem Datensignal gewonnen, mit jeder Stufe nähert sich der VCO der korrekten Frequenz bzw. Phase:

<sup>3</sup>Hintergrund: Die Sync.-Länge wird bestimmt, indem die Anzahl der Nullen hintereinander gezählt wird und daraus das Maximum gebildet wird. Dieser Maximalwert entspricht erst nach Ablauf von mindestens einem Frame der Sync.-Länge. Da die Übertragungsrate bei ADAT zunächst nicht bekannt ist, wird hier keine Zeit abgewartet, sondern eine Anzahl an Transitionen.

**Erste Stufe - 10 Bit Sync** In der ersten Stufe läuft der VCO zunächst mit irgendeiner beliebigen Frequenz. Mit dieser Frequenz wird dann ein Zähler getaktet, der die Anzahl an Nullen hintereinander bestimmt. Bei einer „1“ wird der Zähler zurückgesetzt. Aus allen Zählerständen wird das Maximum gebildet. Nach 256 Piegelwechseln im Datensignal werden Zähler und Maximalwert zurückgesetzt. Wurden mehr als 11 Nullen als Maximum gezählt, ist der VCO zu schnell und seine Frequenz wird verringert. Wurden weniger als 8 Nullen gezählt, ist er zu langsam und wird entsprechend beschleunigt.

Falls keine VCO-Korrektur nötig war, wird Stufe 2 eingeschaltet.

**Zweite Stufe - 256 Bits/Frame** Die zweite Stufe funktioniert im Prinzip genau wie Stufe 1, der einzige Unterschied liegt darin, dass statt aufeinanderfolgenden Nullen der Abstand zwischen den erkannten Synchronisationssequenzen gezählt wird. Liegt der Wert unter 255 wird der VCO beschleunigt, bei mehr als 257 Takten entsprechend die Frequenz verringert.

Falls keine VCO-Korrektur nötig war, geht es weiter zu Stufe 3.

**Dritte Stufe - Phasenkorrektur** In der dritten Stufe wird zunächst geprüft ob die Zähler-Bedingungen aus den Stufen 1 und 2 erfüllt werden: Wird die Bedingung von Stufe 1 erfüllt ( $8 \geq \text{Maximum} \geq 11$ ), wird die Bedingung von Stufe 2 geprüft. Ist eine der Bedingungen nicht erfüllt, so wird sofort in die entsprechende Stufe zurückgesprungen.

Sind beide Bedingungen erfüllt, wird die Phase gemessen und ggf. angepasst. Sind keine Phasenkorrekturen mehr nötig, ist die Phase korrekt und der „Phase locked“-Ausgang wird gesetzt.

Solange die Bedingungen von Stufe 1 und 2 erfüllt sind, wird Stufe 3 wiederholt.

### 2.3.2 Taktgewinnung in einem reinen Digitalsystem

Die für diese Thesis entwickelte Taktgewinnung erfolgt ebenfalls in drei Schritten, die sich allerdings von denen aus dem ADAT-Patent unterscheiden:

1. Erkennung des Synchronisierungssignals
2. Messen der Framelänge
3. Generierung des Bit-Taktes

#### 2.3.2.1 Erkennung der Synchronisationssequenz

Gemäß [Bar+94, Fig.1] ist die Synchronisationssequenz des ADAT-Frames 10 Bit lang. Davor steht aufgrund des Bitstuffings immer eine „1“. Aufgrund der NRZI-

Kodierung (siehe 2.2.4) ergibt sich dabei eine Reihe von 11 Bits, während denen das Signal seinen Pegel nicht ändert<sup>4</sup> (siehe Abbildung 2.5).

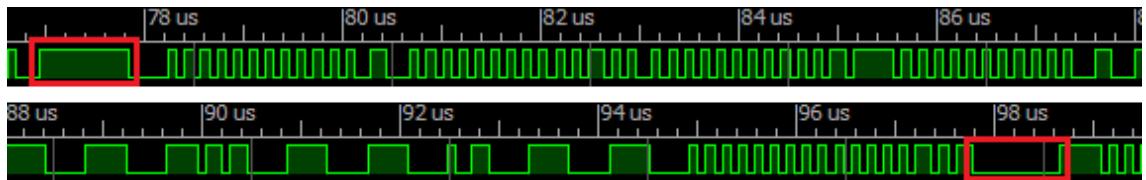


Abbildung 2.4: Die Synchronisationssequenz (Rot) in einem realen ADAT-Signal

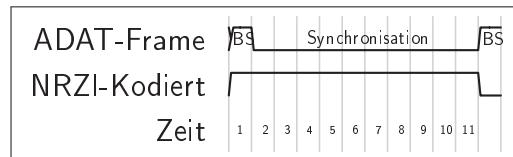


Abbildung 2.5: Synchronisationssequenz NRZI-Kodiert (BS: Bitstuffing)

Die für diese Thesis entwickelte Frame-Sync-Erkennung basiert auf einem (festen) hohen Takt, mit welchem ein Zähler getaktet wird. Genau wie in [Bar+94] beschrieben, wird auch die hier solange gezählt, wie sich das Signal nicht ändert und daraus ein Maximalwert gebildet. Dieser wird nach einer festen Anzahl von Pegelwechseln des Daten-Signals zwischengespeichert. Im Gegensatz zum Verfahren in [Bar+94] wird die Frequenz des daraus generierten Taktes jedoch nicht nur leicht angepasst, sondern jedes mal überschrieben. Dadurch ist der Takt sofort stabil, sobald das Synchronisationssignal einmal korrekt erkannt wurde<sup>5</sup>.

Parallel dazu wird geprüft, ob der Zähler aktuell einen bestimmten Schwellwert, in diesem Falle die Hälfte<sup>6</sup> des Maximalwertes erreicht (oder überschritten) hat. Ist dies der Fall, wurde die Synchronisationssequenz erkannt und eine „1“ wird ausgegeben. Falls der Zählerwert kleiner als der Schwellwert ist, wird der Ausgang auf „0“ gesetzt.

<sup>4</sup>NRZI: Durch die „1“ wird der Pegel geändert, und bleibt für die Dauer eines Bits auf diesem Pegel. Die Nachfolgenden zehn Nullen ändern den Pegel nicht, d.h. es ergeben sich insgesamt 11 gleiche Bits auf der Leitung.

<sup>5</sup>Bei der Methode aus dem ADAT-Patent dagegen nähert sich der VCO in mehreren Schritten dem korrekten Wert an, braucht also länger bis er die korrekte Frequenz erreicht.

<sup>6</sup> $11/2 = 5,5$  Bit, das sind mehr als die normalerweise maximal erreichbaren 5 Bit. Aufgrund der hohen Taktung des FPGA und der damit verbundenen Überabtastung des ADAT-Signals ist eine Erfassung auf ein halbes Bit genau kein Problem.

### 2.3.2.2 Messen der Framelänge

Als nächstes wird die Länge des Frames bestimmt. Dazu läuft wieder ein Zähler mit, der jeweils bei einer fallenden Flanke am Ausgang der Synchronisierungserkennung zurückgesetzt wird. Der Wert des Zählers wird vor dem Zurücksetzen zwischengespeichert (und regelmäßig überschrieben).

### 2.3.2.3 Rekonstruktion des Bit-Taktes

Im FPGA muss die Taktgewinnung digital erfolgen. Ziel ist dabei ein Taktsignal mit einer Flanke, die ungefähr in der Mitte der Datenbits liegt, da nur dort davon ausgegangen werden kann, dass das Datenbit stabil ist. Im Gegensatz zu der Analog-

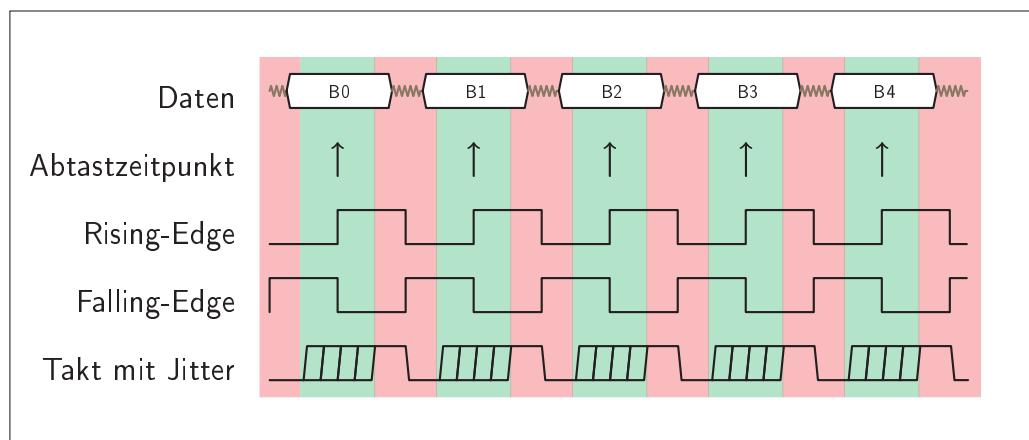


Abbildung 2.6: Abtast-Takt: Das Einlesen der Bits sollte im grünen Bereich erfolgen

variante (siehe 2.3.1), bei der sich der VCO an beliebige Frequenzen anpassen kann, ist bei fest getakteten Digitalsystemen prinzipbedingt keine direkte Ableitung von beliebigen Frequenzen aus dem Taktsignal möglich. Da eine stabile Frequenz des Bit-Taktes allerdings nur im Mittel nötig ist, das Taktsignal aber durchaus einen signifikanten Jitter aufweisen darf (siehe grüner Bereich in Abbildung 2.6) stellt die Erzeugung des Bit-Taktes bei entsprechend hoher Taktfrequenz des FPGAs kein Problem dar.

Aus dem zweiten Schritt ist die Länge eines Frames in Takten bekannt. Zur Taktezeugung kann jetzt der hohe Systemtakt entsprechend heruntergeteilt werden. Dazu wird zunächst die Framedauer durch 512 geteilt – 256 Bits pro Frame; doppelte Frequenz um eine Flanke in der Mitte der Bits zu bekommen – und ein Zähler mit dem resultierenden Wert als Maximum eingerichtet. Läuft dieser Zähler über, so wird das Ausgangssignal des Taktgenerators geändert, es ergibt sich der ADAT-Bit-Takt.

**Phasenkorrektur** Da die Zählerwerte ganzzahlig sind, wäre eine extrem hohe Taktfrequenz notwendig um den Takt bis zum Ende des Frames synchron zu halten, schließlich kann mit jedem Datenbit der Bit-Takt um eine Periode des Systemtaktes abweichen. Um dies zu umgehen wird bei jedem Pegelwechsel auf der Signalleitung der Zähler zurückgesetzt. Falls bereits ein bestimmter Teil der Datenbitdauer abgelaufen ist, wird zusätzlich der Ausgangspegel geändert.

Das Prüfen auf eine Mindestzeit ist dabei nötig, falls der Bitdauer-Zähler einen normalen Bit-Takt-Wechsel erzeugt, kurz danach aber ein Pegelwechsel auf der Datenleitung auftritt (der Bitdauer-Wert also zu klein war). Ohne die Prüfung würde sonst ein Pegelwechsel zu viel erzeugt.

Um den Systemtakt nicht unnötig hoch werden zu lassen, wird noch eine weitere Phasenanpassung vorgenommen: Neben der durch 512 geteilten Framedauer wird auch die durch 256 geteilte Framedauer (d.h. die Taktperiode) gespeichert. Der Zähler für die Taktdauer bekommt als Maximalwert nun die Taktperiode. Bei Erreichen der durch 512 geteilten Framedauer wird (wie bisher) der Ausgang des Taktgenerators geändert. Anstatt jetzt aber den Zähler zurückzusetzen wird einfach weiter gezählt. Erreicht der Zähler die Taktperiode so läuft er über und ändert ebenfalls das Ausgangssignal.

Da der Rundungsfehler der durch 512 geteilten Framedauer sich nun nicht mehr auf alle weiteren Taktflanken fortsetzen kann, sondern nur den Jitter einer der Flanken bestimmt, liegt der maximale Zeitfehler für eine Taktperiode nur noch bei einem statt vorher zwei Bits, d.h. der Fehler hat sich halbiert. Damit kann die Frequenz des Systemtaktes entsprechend geringer gewählt werden.

## 3 McBSP - Die Verbindung mit dem DSP

Die DSPs der TMS3206000<sup>TM</sup>-Familie von TEXAS INSTRUMENTS verfügen unter anderem über eine universell einsetzbare serielle Schnittstelle, den sogenannten McBSP (Multichannel Buffered Serial Port).

Diese Schnittstelle erlaubt Mehrkanal Full-Duplex-Kommunikation über ein einfaches Interface.

### 3.1 Hardware

Hardwareseitig wird eine McBSP-Schnittstelle mit bis zu sieben Pins angebunden:

Jede der zwei Datenrichtungen (Senden/Empfangen) verfügt über je einen Takt, Frame-Sync und Daten-Pin. Außerdem ist es möglich, den Takt für den McBSP wahlweise aus dem DSP-internen Takt oder aus einem externen Takteingang (CLKS) abzuleiten.

Pin-Name	Richtung	Funktion
CLKR	Eingang/Ausgang	Empfang: Takt
FSR	Eingang/Ausgang	Empfang: Frame-Anfang
DR	Eingang	Empfang: Daten
CLKX	Eingang/Ausgang	Senden: Takt
FSX	Eingang/Ausgang	Senden: Frame-Anfang
DX	Ausgang	Senden: Daten
CLKS	Eingang	Externer Takt

### 3.2 Frames, Phasen und Elemente

Die Kommunikation mit dem McBSP erfolgt grundsätzlich Frame-basiert. Dabei kann ein Frame aus bis zu zwei Phasen bestehen. Jede Phase wiederum besteht aus einer bestimmten Anzahl von Elementen mit gleicher Länge

Ein Element ist ein Datenabschnitt einer bestimmten Länge. Die Elementlänge ist nicht frei wählbar: 8,12,16,20,24 oder 32 Bit sind erlaubt. [Tex06b, S. 274]

→ In [Tex06b] wird für „Element“ abweichend der Begriff „Word“ verwendet.

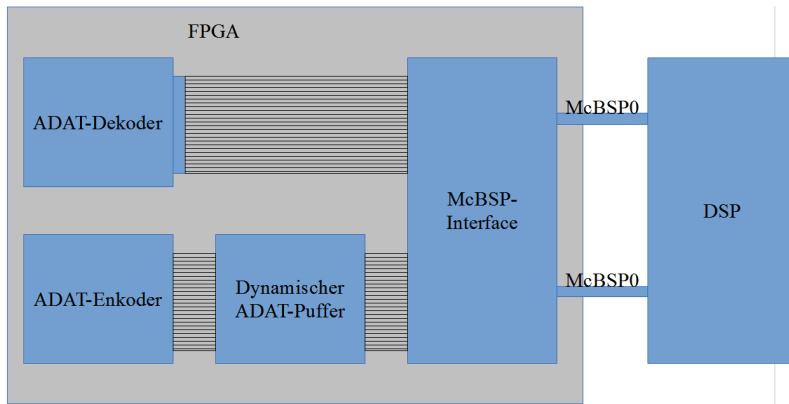


Abbildung 3.1:  
Verbindung des  
FPGAs mit dem  
DSP

### 3.3 Ansteuerung im DSP

*Aufgrund der Tatsache, dass ich bereits durch das Erlernen der Plattform *FPGA* und der Sprache *VHDL* voraussichtlich ausgelastet sein würde, ist die Ansteuerung im *DSP* nicht Teil dieser Thesis und wurde von einer anderen Person bearbeitet. Um sinnvolle Vorgaben für die Kommunikation mit dem *DSP* machen zu können, habe ich mich dennoch ein wenig damit beschäftigt.*

Der McBSP ist eine eigene Funktionseinheit im DSP, die über diverse Kontrollregister des DSPs gesteuert wird. Bei der Ansteuerung gibt es zwei Möglichkeiten:

#### 3.3.1 Ansteuerung: Manuell

Grundsätzlich lassen sich alle Register über ihre jeweilige Adresse direkt erreichen. Dieses Vorgehen ist jedoch wenig intuitiv und wohl vor allem bei der Programmierung in Assembler nötig.

#### 3.3.2 Ansteuerung: CSL

Mit der CSL (Chip Support Library) bietet TEXAS INSTRUMENTS weitaus komfortablere Ansteuermöglichkeiten. Die CSL stellt diverse C-Funktionen für Initialisierung, das Öffnen und Schließen des McBSP-Ports sowie diverse Lese- und Schreibmechanismen für Daten und Kontrollregister zur Verfügung.

Dabei gibt es zwei Ansätze zur Konfiguration der Hardware:

**CSL\_HwSetupRaw** Die Funktion `CSL_HwSetupRaw` erhält die Konfigurationsdaten direkt als Struktur der kompletten Konfigurationsregister, die Konfigurationsbits sind innerhalb der übergebenen Elemente also genau so angeordnet wie im DSP.

**CSL\_McBSPHwSetup** CSL\_McBSPHwSetup dagegen erhält eine thematisch sortierte Struktur, bei der also z.B. alle Einstellungen zum Takt an einer Stelle der Struktur zusammengefasst sind. (*vermutlich die intuitivste Variante*)

→ Weitergehende Informationen finden sich im *TMS320C6455 Chip Support Library API Reference Guide [Tex06b]* im Kapitel 11 „MCBSP MODULE“ ab Seite 253.

### 3.4 Ansteuerung im FPGA

Das gewählte „Protokoll“ zur Kommunikation mit dem McBSP sieht wie folgt aus:

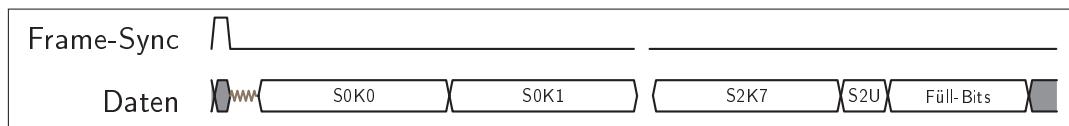


Abbildung 3.2: Frameaufbau McBSP (S0K0: Schnittstelle 0 Kanal 0 S2U: Userbits von Schnittstelle 2)

Alle Daten eines Audio-Samples werden in einem McBSP-Frame übertragen. Nach dem Frame-Sync-Puls am Anfang wird zunächst eine im Code definierbare Zeit abgewartet, bevor die eigentliche Datenübertragung beginnt (siehe Abbildung 3.3).

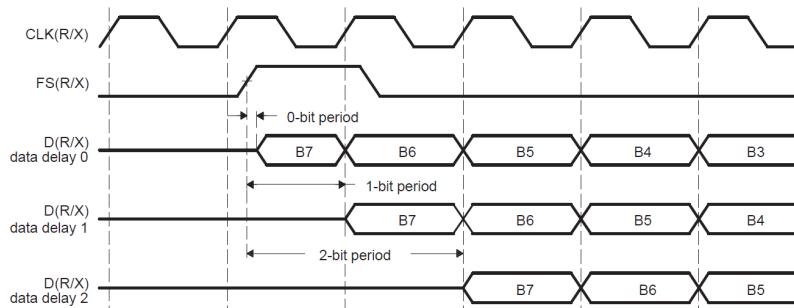


Abbildung 3.3: „Data Delay“ im DSP (Quelle: [Tex06a, S. 28])

Die eigentliche Datenübertragung läuft Schnittstellenweise beginnend mit ADAT-Schnittstelle 0; jeweils erst die Audiodaten (Kanalweise, beginnend bei Kanal 0) und dann die zugehörigen ADAT-Userbits inkl. Füllbits (siehe 3.4.2 Auffüllen auf 24 Bit pro Element). Alle Daten werden mit dem höchstwertigen Bit zuerst übertragen (MSB-first).

### 3.4.1 Warum auch die Userbits?

Die Userbits würden normalerweise für die maximal geforderte (gemäß Besprechung [Kle13]) Abtastrate von 48 kHz nicht benötigt. Aufgrund der Tatsache, dass sie im ADAT-Dekoder bereits empfangen werden und (theoretisch, siehe unten) nur vier Bit pro Schnittstelle belegen, ist es kein großer Aufwand die Userbits ebenfalls durch die gesamte Signalkette mitzunehmen.

Für diesen vergleichsweise kleinen Aufwand erhält man allerdings zahlreiche neue Möglichkeiten beim Ausbau des Projektes. Die mögliche Verwendung von S/Mux war hierbei ausschlaggebend:

- **S/Mux**

Da die eigentliche Datenübertragung bei S/Muxkomplett identisch ist, kann bei Auswertung des S/Mux-Userbits durch den DSP und entsprechender Interpretation der Audiodaten ohne weiteren Aufwand im FPGA auch mit der doppelten Audio-Sample-Rate gearbeitet werden (siehe 2.2.3.1 *S/Mux: Doppelte Abtastrate*).

- **Timecode**

Über Userbit 0 kann ein absoluter Timecode (z.B. für Dokumentationszwecke) übertragen werden.

- **MIDI**

Die Übertragung von MIDI-Daten über Userbit 1 könnte vor allem für allgemeine Steuerdaten genutzt werden.

- **Userbit 3**

Userbit 3 ist reserviert und hat den konstanten Wert „0“. Mit einer entsprechenden Anpassung der zugehörigen Codesegmente könnte hierüber eine interne Datenübertragung zwischen FPGA und DSP erfolgen. Sollen größere Datenmengen übertragen werden, empfiehlt sich allerdings eher die Implementierung einer zusätzlichen Übertragungsphase im FPGA-seitigen McBSP-Interface oder die Nutzung der „Auffüllbits“ (siehe unten).

### 3.4.2 Auffüllen auf 24 Bit pro Element

Da der McBSP zwar diverse verschiedene Elementlängen und auch das Senden von zwei Phasen mit unterschiedlichen Elementlängen unterstützt, ein Senden von lediglich vier Bit pro Element aber nicht möglich ist, werden die Userbits der Einfachheit halber auf 24 Bit (wie bei den Audiodaten) aufgefüllt. Die „Auffüllbits“ werden beim Empfang auf beiden Seiten wieder verworfen.

Dieses Vorgehen ist zwar nicht sonderlich effektiv, ermöglicht aber weiterhin eine problemlose und einfache Skalierung der ADAT-Schnittstellenanzahl. Würden

dagegen die Userbits alle zusammen in einem Element gesendet, so müsste hier je nach Schnittstellenanzahl wieder eine Aufteilung auf mehrere Elemente erfolgen, der Implementierungsaufwand in FPGA und DSP wäre – im Vergleich zur derzeit nötigen Anpassung eines einzelnen Schleifenparameters – also entsprechend höher. Da die Datenraten in diesem Fall mit drei Schnittstellen noch absolut problemlos verarbeitbar sind, wurde auf eine solche Datenoptimierung daher verzichtet.

→ Bei sehr vielen Schnittstellen sollte aufgrund der hohen benötigten Datenrate der McBSP-Leitungen eine effektivere Implementierung in Betracht gezogen werden.

### 3.4.3 Programmablauf im DSP

Ursprünglich war angedacht, dass der DSP als Master arbeitet, die Daten also explizit beim FPGA anfordert, diese dann verarbeitet und, sobald er fertig mit den Berechnungen ist, das Ergebnis zurückschickt.

Letztendlich fiel die Entscheidung aber zugunsten einer „Fließbandarbeiter-Funktion“<sup>1</sup> des DSPs: Der FPGA sendet die Audiodaten, sobald er sie komplett empfangen hat, sofort an den DSP. Dieser berechnet das Gegensignal und schickt es, sobald er fertig ist, ebenfalls sofort zurück.

Dieses Vorgehen hat mehrere Vorteile:

**Kein Sendepuffer im FPGA nötig** Da die empfangenen Daten sofort wieder gesendet werden, ist kein Puffer zwischen dem ADAT-Dekoder und dem McBSP-Sender nötig. Der ADAT-Dekoder legt die neuen Daten erst an seinen Ausgang an, wenn ein kompletter ADAT-Frame empfangen wurde – solange also das Senden der Daten an den DSP schneller erfolgt als das Empfangen der ADAT-Daten, ist ein zusätzlicher Puffer nicht erforderlich.

**Geringere Latenz & Ressourcensparend** Falls der DSP die Daten beim FPGA anfordern würde, ginge hier schon einmal (zugegebenermaßen sehr wenig) Zeit für die Anfrage (= das Frame-Sync-Signal) verloren. Vor allem aber müsste die Anfrage im FPGA auch verarbeitet werden, also zusätzliche Logik implementiert werden.

Im DSP dagegen ist das Erkennen von Frame-Sync schon fest im McBSP eingebaut. So lassen sich unter anderem Interrupts auslösen, wenn eine Aktivität auf der Frame-Sync-Leitung erkannt wird.

Letztendlich wäre ein solches Anfragesystem nicht zielführend, da der FPGA die Daten erst liefern kann, wenn er sie Empfangen hat. Würde der DSP die Daten nun also zu früh oder zu spät anfragen, ginge unnötig Zeit verloren.

<sup>1</sup>Der DSP muss sofort bearbeiten, was ihm gerade „Vor die Nase gesetzt wird“.

Der DSP muss die Daten bereits in Echtzeit verarbeiten können, da das ANC-System ebenfalls in Echtzeit funktionieren soll. Da die Berechnungen im DSP immer gleich lange dauern, kann Frame-Sync eben so gut vom FPGA vorgegeben werden (mit allen entstehenden Vorteilen).

### 3.4.4 Senden

Der Sende-Prozess im FPGA wird vom DSP über dessen CLKR0-Pin getaktet. Frame-Sync dagegen wird im FPGA generiert, was Ressourcen spart, da Frame-Sync nicht mehr erkannt werden muss.

#### 3.4.4.1 Kein Puffer nötig

Das Modul vor dem McBSP-Sender, der ADAT-Dekoder, besitzt bereits einen „Pseudo-Puffer“: Sein Ausgang wird erst dann geändert, wenn der gesamte Frame empfangen wurde. Solange nun also die Daten wie oben bereits erwähnt vom McBSP-Sendemodul schneller gesendet werden als der ADAT-Dekoder den nächsten Frame empfangen werden, ist problemlos auch ein direkter Zugriff auf den Ausgang des ADAT-Dekoders möglich. Ein Eingangspuffer beim Sender ist folglich unnötig, was weitere Ressourcen einspart.

#### 3.4.4.2 Steuerung über eine Statemachine

Sobald über den Eingang Neue\_Daten\_Zum\_Senden erkannt wird, dass neue Daten an den DSP gesendet werden sollen, wird der Frame-Sync-Pin auf den entsprechenden Aktiv-Pegel FSX\_Aktiv gesetzt und die Sende-Statemachine wechselt von der Phase WartenAufDaten nach FrameAnfangAbwarten, es sei denn die Wartezeit FSX\_Verzögerung ist 1 (sonst wird direkt nach ADAT\_AudioDaten gewechselt). Hier wird nun für eine entsprechende Anzahl an Takten (siehe Abbildung 3.3) gewartet; dann wechselt die Statemachine in die Phase ADAT\_AudioDaten. Sobald eine der Phasen ADAT\_AudioDaten oder FrameAnfangAbwarten betreten wird, wird das Frame-Sync-Signal wieder auf FSX\_Inaktiv gesetzt.

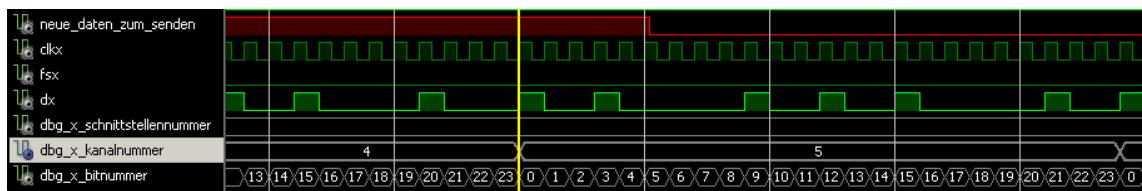


Abbildung 3.4: Simulation des Sendevorgangs vom FPGA zum DSP - hier in der Phase ADAT\_AudioDaten

In der Phase ADAT\_AudioDaten werden nun mit jedem Takt entsprechend der aktuell zu sendenden Daten die Variablen Schnittstellennummer, Kanalnummer und Bitnummer weiter gezählt (Abbildung 3.4) und das durch diese Variablen adressierte Bit ausgegeben. Sobald die Audio-Daten für eine ADAT-Schnittstelle gesendet wurden, wechselt die Statemachine nach ADAT\_Userbits, sendet nach dem selben Prinzip die Userbits und wechselt wieder zurück zu ADAT\_AudioDaten.

Sobald alle Daten gesendet wurden, d.h. die nun aktuelle Phase sollte ADAT\_Userbits sein und die Schnittstellennummer hat ihrem Maximalwert erreicht, wird wieder in die Phase WartenAufDaten gewechselt.

Um auch nach einer Störung eine korrekte Übertragung der nachfolgenden Frames zu ermöglichen, werden bei einem Frame-Sync-Puls alle internen Zähler (s.o.) auf Null zurückgesetzt.

### 3.4.5 Empfangen

Auch das Empfangsmodul wird durch eine Statemachine gesteuert. Zunächst befindet diese sich in der Phase WartenAufFrameSync. Sobald ein aktives Frame-Sync-Signal erkannt wird, geht das Empfangsmodul in die Phase FrameAnfangAbwarten über, sofern die Frame-Sync-Verzögerung nicht auf „0“ eingestellt ist (sollte dies der Fall sein, wechselt die Statemachine direkt in die Phase ADAT\_AudioDaten).

In der Phase FrameAnfangAbwarten wird die eingestellte Verzögerung zwischen Frame-Sync und dem Anfang der Daten abgewartet (vgl. Abbildung 3.3 in Abschnitt 3.4 Ansteuerung im FPGA). Sobald diese Zeit abgelaufen ist, wird auch von hier in die ADAT\_AudioDaten-Phase gewechselt.

Nun werden die Audiodaten der ersten Schnittstelle übertragen. Danach wechselt die Statemachine in die Phase ADAT\_Userbits, in der die Userbits übertragen werden. Dieser Vorgang wiederholt sich nun für alle aktuell eingestellten ADAT-Schnittstellen; danach wechselt die Statemachine wieder in die „Standby“-Phase WartenAufFrameSync.

#### 3.4.5.1 Details der Implementierung

Ein 24-Bit FiFo-Puffer<sup>2</sup>, der um ein Zugriffsmöglichkeit auf alle Bits gleichzeitig ergänzt wurde, liest mit jedem Takt an CLKR ein Bit ein, und zwar unabhängig von der aktuellen Empfangsphase. Eine Berücksichtigung der aktuellen Statemachine-Phase wäre zwar möglich, würde aber den Hardwareaufwand im FPGA nur erhöhen, da die vorhandene FiFo-Hardware um eine Deaktivierungsmöglichkeit wie z.B. einen Clock-Enable-Eingang ergänzt werden müsste. Andererseits kostet es im FPGA nichts, das FiFo permanent laufen zu lassen. Es muss lediglich sichergestellt

---

<sup>2</sup>FiFo: First in-First out - Wie der Name schon verrät, gibt dieser Puffer das zuerst hineingelegte Element auch als erstes wieder aus.

werden, das zu jeder Zeit klar ist, ob und wo sich relevante (also zu empfangende) Daten im FiFo befinden.

Diese Information, also das Herzstück des eigentlichen Empfangsprozesses, wird durch die jeweils aktuelle Phase und die drei Variablen Schnittstellennummer, Kanalnummer und Bitnummer gesteuert. Beim Übergang in eine neue Phase bzw. nach dem Empfang einer kompletten Dateneinheit werden diese Variablen immer angepasst. So wird z.B. in der Phase ADAT\_AudioDaten die Kanalnummer erhöht, sobald ein Kanal komplett empfangen wurde, die Bitnummer also ihren jeweiligen Maximalwert erreicht hat (Dateneinheit komplett). Die nun vollständige Dateneinheit wird dann dem Ausgang zugewiesen und die nächste Einheit kann empfangen werden.

Sind alle acht Audiokanäle vollständig empfangen worden, wird in die nächste Phase (ADAT\_Userbits) gewechselt. Hier beträgt der Maximalwert der Bitnummer genau wie bei den Audiodaten 23 (24 Bit: es wird von 0 bis 23 gezählt). Zunächst werden die vier Userbits empfangen; danach wird abgewartet, bis ein komplettes 24-Bit Element voll ist. Dementsprechend wird nur der relevante Teil des FiFos ausgewertet.

### 3.4.5.2 Der goldene Mittelweg zwischen Puffer und Multiplexer

Mit der Verwendung eines 24-Bit-Puffers geht auch die automatische Generierung eines 24-Bit 8-zu-1-Multiplexers für die Kanäle im Frame, und z.B. bei 3 ADAT-Schnittstellen je eines 192-Bit- und eines 4-Bit 3-zu-1-Multiplexers für Audio bzw. Userbit-Daten pro Schnittstelle durch das Synthesewerkzeug einher. Da die Anbindung des Ausgangs und die entsprechende Weiterverarbeitung der Daten mit der Wortbreite der gesamten mit einem Audiosample verbundenen Daten erfolgt, liegt es eventuell auch nahe, den Puffer im Empfangsteil des McBSP genauso breit zu implementieren und den gesamten McBSP-Frame ohne das lästige Mitzählen von Schnittstellen- und Kanalnummern und diverse Phasenwechsel zu erledigen, indem einfach ein entsprechend großes FiFo und ein Bitzähler (der zählt, bis alle Bits eines Frames empfangen sind) den Empfang übernehmen würden.

Ein solches Vorgehen ist allerdings aufgrund der enormen Speicherbreite an dieser Stelle nicht zu empfehlen und sollte nur genutzt werden, wenn es nicht anders geht. In diesem Fall kann davon ausgegangen werden, dass die Daten von der nächsten Stufe gepuffert werden, ein eigener Puffer wäre somit nicht nötig.

Der 24-Bit Puffer (Die Größe entspricht den Daten von einem Audio-Kanal) bietet hier einen guten Kompromiss zwischen einem gigantisch breiten Puffer von z.B. 588 Bit (bei 3 ADAT-Schnittstellen) und einem entsprechend aufwendigen N-zu-1-Multiplexer (wie er bei direktem Zuweisen aller Werte ohne Puffer nötig wäre). Ziel hierbei ist es, sowohl Speicherelemente als auch normale Logik gleichermaßen zu verwenden, um den FPGA insgesamt möglichst gleichmäßig zu füllen.

### 3.4.5.3 Alle Daten sind vollständig

Sobald der gesamte McBSP-Frame empfangen wurde, wechselt das Empfangsmodul (wie bereits erwähnt) wieder in die WartenAufFrameSync-Phase. Gleichzeitig ändert der 1-Bit-Ausgang Daten\_komplett\_empfangen seinen Wert, um dem nachfolgenden Modul damit zu signalisieren, dass die Daten nun komplett sind und abgeholt werden können.

## 4 Realisierung in einem FPGA

Als Entwicklungsumgebung wurde ISE von XILINX in der Version 14.6 verwendet. Alle Simulationen erfolgten mit dem mitgelieferten Programm ISim.

### 4.1 Wahl des konkreten FPGAs

Die ersten Tests auf Hardware erfolgten mit dem „Spartan-3A FPGA Starter Kit“ von XILINX. Auf diesem ist ein Spartan-3A-XC3S700A im BGA (Ball Grid Array)-Gehäuse verbaut.

Die FPGAs der Spartan-3A-Serie werden in sieben verschiedenen Gehäuseformen angeboten, wobei nicht jeder FPGA auch in jedem Gehäuse erhältlich ist:

Bis auf VQ100 und TQ144 sind alle Gehäuseformen BGA. Da BGA-ICs die Kontakte unterhalb des ICs haben, ist ein Löten von Hand nicht möglich. Auch beim Reflow-Löten<sup>1</sup> ist eine Kontaktkontrolle hilfreich, welche z.B. durch Röntgen der Platine erfolgen kann. Insgesamt bedeutet die Verwendung eines ICs im BGA-Gehäuse für Einzelstücke einen enormen Zusatzaufwand bei der Bestückung.

Aus diesem Grund war eines der Ziele, möglichst einen FPGA mit einem „normalen“ Gehäuse (also einem mit den typischen „IC-Beinchen“) zu verwenden. Damit

<sup>1</sup>Reflow-Löten: Auf sämtliche Pads der Platine wird eine Lötpaste aufgetragen, danach werden die Bauteile darauf platziert. Das Löten erfolgt nun indem ein (für kritische Bauteile in der Regel im Datenblatt angegebenes) Temperaturprofil in einem Reflow-Ofen durchfahren wird. Dabei schmelzen die in der Lötpaste enthaltenen Lötzinnkügelchen und verlöten das Bauteil.

Gehäuse	VQ100	TQ144	FT256	FG320	FG400	FG484	FG676
XC3S50A	✓	✓	✓	✗	✗	✗	✗
XC3S200A	✓	✗	✓	✓	✗	✗	✗
XC3S400A	✗	✗	✓	✓	✓	✗	✗
XC3S700A	✗	✗	✓	✗	✓	✓	✗
XC3S1400A	✗	✗	✓	✗	✗	✓	✓

Tabelle 4.1: Gehäuseformen der Spartan-3A-Serie nach [Xil10b, S.5, Tabelle 2]

Die Namen der Gehäuseformen sind wie im Datenblatt verkürzt angegeben, z.B. VQ100 → VQFP-100

bleiben nur noch der XC3S50A und der XC3S200A, welche sich hauptsächlich in der Menge der verfügbaren Logikzellen unterscheiden (siehe [Xil10b, S.3, Tabelle 1]). Da der große XC3S700A zu dem Zeitpunkt, als es an den Platinenentwurf ging, trotz einiger Optimierungen noch ca. zu 50 % gefüllt war, stand trotz der Aussicht auf mögliche weitere Verbesserungen schon früh fest, dass der XC3S50A auf jeden Fall zu klein sein würde.

Damit fiel die Wahl auf den XC3S200A in VQFP-100. Aufgrund der geringen Preisdifferenz wurde die schnellere Variante (der sogenannte „Speedgrade 5“) bestellt.

## 4.2 Allgemeines zu VHDL

*Dieser Abschnitt gibt eine kurze Übersicht über die wichtigsten in dieser Thesis verwendeten VHDL-Konstrukte.*

VHDL ist nicht case-sensitiv, unterscheidet also nicht zwischen **entity** und **Entity**.

### 4.2.1 Entity

Die **Entity** stellt die Schnittstelle eines VHDL-Moduls zur Außenwelt dar. In [RS13] wird sie mit einem IC-Gehäuse verglichen:

In der mit **entity** bezeichneten Entwurfseinheit werden die Schnittstellen eines VHDL-Funktionsblocks nach außen beschrieben. In einem Vergleich eines Board-Designs mit einem VHDL-Quellcode stellt die **entity** den zu bestückenden IC-Gehäusetyp dar, der durch die Anzahl und die Bezeichnung der Anschlüsse eindeutig definiert ist. [RS13, S. 7]

Diese Beschreibung ist zwar eingängig, scheint aber zu kurz gegriffen, da sich die Anzahl der Eingänge bei entsprechendem VHDL-Code über **generics** beliebig einstellen lässt, was bei ICs (trotz dem möglichen Wechsel zu einer anderen Gehäuseform) in der Regel in so großem Umfang nicht mehr möglich ist.

Eine Entity wird folgendermaßen definiert:

```

1 entity Addierer is
2   generic (Bits : natural range 1 to 32 := 8);
3   port(ZahlA : in bit_vector(Bits-1 downto 0);
4     ZahlB : in bit_vector(Bits-1 downto 0);
5     Ergebnis : out bit_vector(Bits-1 downto 0) );
6 end Addierer;

```

Listing 4.1: Entity-Beispiel

#### 4.2.1.1 Generics

Mit **generic** (`Bits : natural range 1 to 32 := 8`); wird hier ein Generic-Parameter angegeben. Dieser kann in der **Architecture**, aber auch schon in der **port**-Anweisung der **Entity** verwendet werden und verhält sich wie eine Konstante. Der Initialisierungswert des Parameters entspricht dem Standardwert – dieser ist nötig, da **generic**-Parameter bei der Instanziierung (siehe 4.2.5.1 *Instanziierung*) nicht angegeben werden müssen.

In diesem Beispiel kann bei der Instanziierung über diesen Parameter später die Anzahl der Bits des Addierers angepasst werden.

#### 4.2.1.2 Ports

Über die **port**-Anweisung werden die tatsächlichen Datenleitungen zur Außenwelt erstellt. In diesem Fall heißen diese `zahlA`, `zahlB` und `Ergebnis` und sind alle vom Typ `bit_vector`, wobei die Breite nicht festgelegt ist sondern über den **generic**-Parameter `Bits` bestimmt wird.

Mit **in** und **out** wird angegeben, ob es sich um einen Eingang oder einen Ausgang handelt. Zusätzlich gibt es noch die Varianten **inout** und **buffer**. **inout** ermöglicht eine bidirektionale Kommunikation, **buffer** ist ein Ausgang, dessen Wert intern auch gelesen werden kann.

Laut P.J.Ashenden kann zwar intern auch von einem **out**-Port gelesen werden, allerdings wird dies nach Ashenden meist nur für die interne Verifikation der Ausgangswerte benutzt, während **buffer** verwendet wird, wenn der zurück gelesene Wert für die tatsächliche Funktion des Moduls verwendet wird[Ash08, S. 138].

→ Sowohl bei **generic** als auch bei **port** wird das Semikolon am Ende der Zeile beim letzten Parameter weggelassen

#### 4.2.2 Architecture

Wenn die **Entity** das Gehäuse des ICs ist, so ist die **Architecture** das Innenleben. Die Architecture enthält die Funktionsbeschreibung/das Verhalten des VHDL-Moduls. Definiert wird sie wie folgt:

```

1 architecture Verhalten of Addierer is
2     --Signale, Konstanten, Komponenten
3 begin
4     --VHDL-Code
5 end Addierer;

```

Listing 4.2: Architecture-Beispiel

Signale, Konstanten und Komponenten (siehe 4.2.5 *Komponenten*), die im VHDL-Code verwendet werden sollen, können zwischen **is** und **begin** eingetragen werden. Einer Entity können mehrere Architectures zugewiesen werden.

### 4.2.3 Signale, Variablen und Konstanten

Signale werden mit **signal** Signalname : Datentyp; deklariert. Sie sind wie elektrische Verbindungspunkte zu verstehen. Einem Signal kann mit Signalname <= Wert; ein Wert zugewiesen werden.

Variablen werden mit **variable** Variablenname : Datentyp; deklariert und sind nur in Prozessen (siehe unten) lokal gültig. Der Zuweisungsoperator sieht hier anders aus: Variablenname := Wert;

Konstanten werden mit **constant** Konstantenname : Datentyp := Wert angelegt.

#### 4.2.3.1 Initialisierungsmöglichkeiten

Weder Variablen noch Signale müssen in VHDL zwingend initialisiert werden, für die Verständlichkeit des Codes ist es allerdings hilfreich. Konstanten dagegen bekommen ihren Wert naturgemäß immer bei der Deklaration (siehe oben).

Die Initialisierung einer Variablen erfolgt als Deklaration mit einer direkten Zuweisung:

```
1 Variable Variablenname : Typ := Wert;
2 Signal     Signalname      : Typ := Wert;
```

Listing 4.3: Initialisierung von Variablen und Signalen

Die Zuweisung des Initialisierungswertes eines Signals erfolgt ebenfalls mit dem Operator := .

#### 4.2.3.2 Datentypen

Datentyp	Erklärung	Beispielwerte
bit	Ein Bit	0; 1
std_logic	Logikpegel	U; X; 0; 1; Z; W; L; H
bit_vector	Bit-Array	"000101"; "111100"
std_logic_vector	Logikpegel-Array	"01ZZ1H0L", "SZZZXLUWH"
integer	vorzeichenbehaftete Ganzzahl	42; 0; -21; 1023
natural	positive Ganzzahl	0; 3; 17; 42

Tabelle 4.2: Einige Datentypen

Signale, Variablen und Konstanten müssen jeweils einem Datentyp zugeordnet werden. Die Standarddatentypen umfassen (unter anderem) die in Tabelle 4.2 aufgeführten Typen `bit`, `std_logic`, `bit_vector`, `std_logic_vector`, `integer` und `natural`. Hierbei nehmen die `_vector`-Typen eine besondere Stellung ein: Sie entsprechen einem Array von Werten des vor `_vector` stehenden Datentyps und benötigen eine `range`-Angabe, mit der die Anzahl der Array- bzw. Vektorelemente festgelegt wird. Die Reihenfolge der Indizes der Array-Elemente hält sich dabei genau an die `range`-Angabe.

#### 4.2.3.3 Range

Mit `range` wird ein Wertebereich angegeben, wie er z.B. für die Angabe der Elementanzahl in einem Array benötigt wird. Außerdem kann damit auch der Wertebereich eines Datentyps eingeschränkt werden:

```
1 signal RaederAmAuto : natural range 0 to 4 := 4;
2 variable Liste Von Bits : bit_vector(10 downto 3);
```

Listing 4.4: Die Range-Angabe

Wie oben zu sehen ist, muss eine `range` weder bei „0“ beginnen noch aufhören. Sie kann entweder in aufsteigender (1 `to` 4) oder in absteigender Reihenfolge (25 `downto` 3) vorliegen. Wenn sie die Elemente eines Arrays angibt, wird das Schlüsselwort `range` weggelassen und die Bereichsangabe in runde Klammern gesetzt (vgl. Zeile 2). Neben den hier gezeigten Ganzzahlen ist z.B. auch die Verwendung von Enumerationstypen möglich.

#### 4.2.3.4 Enumerationstypen, Arrays & eigene Datentypen

Enumerationstypen eignen sich vor allem für die automatische Zuordnung (in dieser Zuordnung) eindeutiger Werte zu Namen, die ansonsten fehleranfällig einzeln als Konstanten mit dem entsprechenden Wert definiert werden müssten, obwohl es eigentlich nicht darauf ankommt, welchen Wert welche Konstante hat, solange die Werte nur eindeutig sind.

```
1 type Übertragungsphase is (WartenAufDaten,
                           FrameAnfangAbwarten, ADAT_AudioDaten, ADAT_Userbits,
                           Benutzerdaten);
2 type Audio8x24 is array (0 to 7) of bit_vector(23 downto 0)
   ;
3 type IntegerArray is array (natural range <>) of integer;
4 variable MeinArray : IntegerArray(1 to 10);
```

Listing 4.5: Typdefinitionen und Enumerationstypen

In VHDL werden sie als neuer Typ definiert, der anschließend für die Variablendefinition verwendet werden kann.

Genauso erfolgt auch das Erstellen von Arrays: Zunächst muss ein entsprechender Arraydatentyp definiert sein (vgl. Listing 4.5, Zeile 2), dann kann z.B. eine Variable dieses Typs deklariert werden.

Neue Datentypen werden in VHDL mit dem **type**-Befehl definiert (**type** Datentypname **is** DefinitionDesDatentyps;). **range**-Angaben müssen hier nicht zwingend als konkrete Werte angegeben werden, sondern können auch offengelassen werden, indem die Bereichsangabe durch Datentyp **range** <> ersetzt wird (Datentyp ist hierbei der Typ mit dem nachher die Grenzen der Bereichsangabe angegeben werden müssen). Wird der Bereich offengelassen spricht man von sogenannten „Unconstrained“-Typen, also Typen, die noch nicht endgültig definiert sind. Bei der Deklaration von z.B. Variablen einer solchen Datentyps muss daher der weggelassene Bereich zwingend angegeben werden (vgl. Listing 4.5, Zeile 4).

#### 4.2.4 Prozesse

Normalerweise werden in VHDL alle Befehle gleichzeitig ausgeführt. Prozesse dagegen erlauben die sequentielle Ausführung von Anweisungen. Ein Prozess kann wie alle VHDL-Anweisungen nur im Code-Block der **architecture** stehen. Mit einem vorangestellten Prozessname: kann ihm ein Name zugewiesen werden, dieser ist aber nicht zwingend erforderlich.

```

1 OptionalerProzessName: process (Sensitivitätsliste)
2      VariablenDeklarationen
3 begin
4      Anweisungen
5 end process;
```

Listing 4.6: Prozesse in VHDL

Die Sensitivitätsliste enthält eine Liste von Signalen. Sobald sich eines der Signale ändert, wird der Prozess aufgerufen.

→ An dieser Stelle ein wichtiger Hinweis: Bei der Synthese wird die Sensitivitätsliste in der Regel komplett verworfen und aus allen Signalen, auf die innerhalb des Prozesses zugegriffen wird, neu erstellt. Für die Simulation dagegen ist eine korrekte Sensitivitätsliste essentiell.

Vor dem **begin** können nun Variablen deklariert werden. Diese sind nur lokal im Prozess gültig.

Typischerweise erfolgt nach **begin** direkt eine Abfrage auf eine bestimmte Änderung der Signale in der Sensitivitätsliste. Dabei gibt es zwei Varianten: Flanken in std\_logic-Signalen werden in der Regel mit **if rising\_edge(Signalname)** **then** bzw. **if falling\_edge(Signalname)** **then** abgefragt. Diese Funktionen bieten erweiterte Flankenerkennungsmechanismen, die die zusätzlichen Logikpegel von std\_logic auswerten können.

Alternativ kann eine mehr oder weniger vergleichbare Funktionalität auch mit **if Signalname'event and Signalname=Vergleichswert** erzielt werden. Für die Erkennung der steigenden Flanke im Signal clk würde man hier also **if clk'event and clk='1' then** schreiben. Signalname'event wird wahr, sobald sich das Signal ändert. Eine Abfrage mehrerer gleichzeitiger Signaländerungen wie z.B. **if SignalA'event and SignalB'event and Irgendeinwert='1'** ist nicht erlaubt.

Die Befehle im Prozess werden daraufhin sequentiell abgearbeitet. Das getaktete Verhalten des Prozesse führt dazu, dass der einem Signal mit Signalname <= Wert; zugewiesene Wert erst am Ende des Prozesses übernommen wird. Dies ist vor allem zu Bedenken, wenn der Wert des Signals im Verlauf des Prozesses auch gelesen werden soll.

### 4.2.5 Komponenten

Soll eine Funktion, die eine Entity zur Verfügung stellt, an einer anderen Stelle verwendet werden, so muss diese im Deklarationsblock der **Architecture** zunächst mit **component** als Komponente deklariert werden. Alle Generics und Ports sind hier entweder genauso wie in der ursprünglichen Entity aufzuführen oder wegzulassen.

```

1 component Addierer
2   generic(Bits : natural range 1 to 32 := 8);
3   port(ZahlA : in bit_vector(Bits-1 downto 0);
4     ZahlB : in bit_vector(Bits-1 downto 0);
5     Ergebnis : out bit_vector(Bits-1 downto 0) );
6   end component;
```

Listing 4.7: Komponentendeklaration

Auch wenn die Entity aus einer anderen Quellcode-Datei kommt, ist kein Include-Statement nötig (wie es z.B. in C/C++ erforderlich ist), solange alle Dateien zum selben Projekt gehören.

#### 4.2.5.1 Instanziierung

Um die Komponente nun zu verwenden, muss diese (irgendwo im ganzen normalen VHDL-Code der übergeordneten **Architecture**) instanziert werden.

```

1 Architecture BesondersSchnell of Taschenrechner is
2     signal ErsteZahl : bit_vector(3 downto 0);
3     signal ZweiteZahl: bit_vector(3 downto 0);
4     signal Resultat   : bit_vector(3 downto 0);
5 begin
6     inst_Addierer: Addierer
7         generic map(Bits => 4);
8         port map(ZahlA => ErsteZahl ,
9                     ZahlB => ZweiteZahl,
10                    Ergebnis => Resultat );
11     ...weiterer Code...

```

Listing 4.8: Instanzierung von Komponenten

Wie bei einem Prozess lässt sich auch hier optional mit `Instanzname:` eine Name für die Instanz festlegen. Danach können mit `generic map` den möglicherweise vorhandenen generic-Parametern neue Werte zugewiesen werden, sofern erwünscht.

Den einzelnen Port-Signalen kann im `port map`-Abschnitt mit `Originalname => ZugewiesenesSignal` ein Signal aus der `architecture` oder der dazugehörigen `entity` zugewiesen werden.

Soll ein Port-Signal zwar aufgeführt, aber nicht verbunden werden, so ist dies mit `=> open` möglich. Alternativ kann man das Signal auch einfach in der `port map`-Auflistung weglassen.

#### 4.2.6 Generate

Mit `generate` können beliebige VHDL-Konstrukte dynamisch erzeugt werden, die sonst jedes mal mühsam von Hand kopiert und angepasst werden müssten. Die Syntax basiert dabei auf einer for-Schleife:

```

1 RAM_Verdrahten: for ADAT_Kanal in 0 to PUFFERBREITE-1
2     generate
3         --Signale:
4         Ausgang(ADAT_Kanal).Kanaele(0) <= BR_do(
5             ADAT_DATENMENGE*ADAT_Kanal +0*24 +23 downto
6             ADAT_DATENMENGE*ADAT_Kanal +0*24 );
7     ...
8 end generate;

```

Listing 4.9: Der „Generate“-Befehl

In Listing 4.9 ist ein Ausschnitt aus dem Modul `DynamischerADATPuffer_BRAM` zu sehen. Der `generate`-Befehl wird hier verwendet um die Ausgänge eines mit

**generic** parametrisierten RAMs an eine ebenfalls mit **generic** erzeugte Datenstruktur anzubinden. Der Parameter ADAT\_Kanal kann hier, genau wie bei **generic**, wie eine Konstante verwendet werden.

#### 4.2.7 Packages

Häufig benutzte Konstanten, Datentypen und Funktionen können in sogenannten „Packages“ zusammengefasst werden:

```

1 package Packagename is
2     Konstanten, Typdefinitionen, Funktionsdeklarationen.
3 end Packagename;
4 package body Packagename is
5     Funktionsimplementierung
6 end Packagename;
```

Listing 4.10: Packages in VHDL

Mit **package** wird ein neues Package erstellt. Konstanten und Typdefinitionen können direkt in dem Bereich in Zeile 2 angegeben werden. Sofern auch Funktionen beteiligt sind, werden von diesen zunächst nur die entsprechenden Deklarationen angegeben. Der Package-Bereich ist also vergleichbar mit den Header-Dateien von C/C++.

Im **package body** werden dann die Funktionen implementiert (Vergleichbar mit den \*.c bzw \*.cpp-Dateien bei C/C++).

Eingebunden werden Packages mit **use**.

```
1 use work.Packagename.all;
```

In diesem Fall werden alle Elemente des Packages eingebunden (**.all**). Alternativ können auch einzelne Elemente mit **.Elementname** explizit herausgesucht werden.

#### 4.2.8 Testbenches

Testbenches sind spezielle VHDL-Module, die zum Testen des VHDL-Codes in der Simulation gedacht sind (siehe ?? ??). Sie bestehen aus einer komplett leeren Entity, was darin begründet liegt, dass sie nie in andere Module eingebunden werden, sondern in der Simulation immer als Top-Modul gelten.

Ansonsten unterscheiden sie sich lediglich in den verwendbaren VHDL-Befehlen von normalen Modulen, die in der Regel schließlich irgendwann synthetisiert werden sollen (siehe 4.4.1 *Nicht synthetisierbare Konstrukte*).

So ist z.B der **after**-Befehl nur in Testbenches zulässig, da er nicht synthetisierbar ist. Mit **after** lassen sich zeitliche Verzögerungen realisieren:

```

1 EinSignal <= '1', '0' after 3 ms, '1' after 2 ms;
2 NochEinSignal <= '1', '0' after 3 ms;
```

Listing 4.11: Zeitliche Verzögerung mit **after**

Die obige Anweisung setzt EinSignal nacheinander mehrmals auf „1“ und „0“. Die Zeitangaben sind dabei absolut zu sehen, die zweite Zeile erzeugt also den selben Signalverlauf an NochEinSignal (die doppelten Zuweisungen schon gesetzter Werte bei richtiger zeitlicher Sortierung wurden hier weggelassen). Die verzögerte Zuweisung verzögert dabei nicht die Ausführung der nächsten Zeile.

Eine andere Wartemöglichkeit ist **wait for** Zeitangabe;. Hier können wie auch bei **after** beliebige Zeitwerte wie z.B. 42 ps eingetragen werden.

fs	ps	ns	us	ms	sec	min	hr
1000 fs	1000 ps	1000 ns	1000 us	1000 ms	1000 sec	60 min	60 hr

Tabelle 4.3: Vordefinierte Zeit-Suffixe in VHDL nach [Ash08, S. 42]

#### 4.2.9 Das Programm „Tektronix CSV -> VHDL Testbench“ - Verwenden von real aufgezeichneten Daten für die Simulation

Weil die Pegel der anfangs für Messungen am Oszilloskop verwendeten ADAT-Empfangsbuchse TORX173 nicht mit den 3,3 V des FPGAs übereinstimmten und ein praktischer Test des Empfangsprogrammes daher nicht möglich war, mussten zunächst sämtliche Tests in Bezug auf den Empfang von ADAT-Daten in der Simulation ablaufen.

Aufgrund der komfortablen Möglichkeit, kleine Programme inklusive Benutzeroberfläche extrem schnell zu erstellen, wurde dieses Programm in Visual Basic 5 geschrieben. Neuere Versionen von Visual Basic ermöglichen in ersten Tests deutliche Geschwindigkeitsgewinne. Da der Simulator ISim mit den großen Dateien, bei denen der Geschwindigkeitsgewinn relevant wird, bereits nicht mehr zurecht kommt, wurde auf eine vollständige Portierung in eine neuere Version letztendlich verzichtet.

Da das Simulieren eines VHDL-Moduls mit einer selbstgeschriebenen Testbench, welche die eventuelle Probleme eines realen elektrischen Signals gar nicht aufweist, bei weitem nicht so aussagekräftig wie ein wiederum nicht immer möglicher Test mit realer Hardware ist, soll hier ein Zwischenweg beschrieben werden:

Die meisten Speicheroszilloskope besitzen eine Möglichkeit, die aufgezeichneten Daten auf externe Datenträger zu speichern. Da die exportierten Dateien nun aber in einem vom Hersteller abhängigen Dateiformat vorliegen, müssen zunächst in VHDL-Code konvertiert werden, bevor sie als Daten in einer Testbench verwendet werden können.

Das in diesem Fall verwendete TEKTRONIX DPO4034 bietet unter anderem den Export als CSV-Datei an:

```

1 Model,DPO4034
2 Firmware Version,2.15
3
4 Point Format,Y
5 Horizontal Units,S
6 Horizontal Scale,4e-06
7 Sample Interval,4e-10
8 Record Length,100000
9 Gating,0.0% to 100.0%
10 Probe Attenuation,10
11 Vertical Units,V
12 Vertical Offset,0

```

```

13 Vertical Scale,2
14 Label,
15 TIME,CH1
16 -2.52800e-05,0.24
17 -2.52796e-05,0.24
18 -2.52792e-05,0.24
19 -2.52788e-05,0.24
20 -2.52784e-05,0.24
21 -2.52780e-05,0.16

```

Listing 4.12: CSV-Datei des Tektronix DPO4034-Oszilloskops

Wie leicht zu erkennen ist, beschreiben die ersten beiden Zeilen das verwendete Oszilloskop. Darauf folgen nach einer Leerzeile die Einstellungen zum Zeitpunkt der Datenaufnahme und weitere zur Interpretation der Daten erforderliche Informationen wie z.B. die Einheiten der horizontalen bzw. vertikalen Achse. Zeile 15 gibt den Namen des exportierten Kanals an, in diesem Falle CH1.

Ab Zeile 16 folgen nun die aufgenommenen Daten, wobei links immer der aktuelle Zeitstempel steht (in diesem Falle beginnt er negativ) und rechts der dazugehörige Messwert.

#### 4.2.9.1 Informationsextraktion aus der CSV-Datei

Die Informationsextraktion aus der CSV-Datei erfolgt durch einen einfachen Parser, welcher nach den in Listing 4.12 blau markierten Zeichenfolgen sucht. Hat er nun einen der Parameter gefunden (z.B. *Vertical Units*) so geht er eine Position weiter (das Komma) und liest ab diesem Punkt bis zum Ende der Zeile. Da anhand des Parameternamens auch der dazugehörige Datentyp bekannt ist, muss der Wert nur noch durch eine entsprechende Datentypkonvertierung interpretiert werden.

Der Anfang der Daten wird anhand des Schlüssels „*Label*,“ gefunden. Von dort aus werden zwei folgende Zeilenumbrüche gesucht, nach denen direkt die Daten

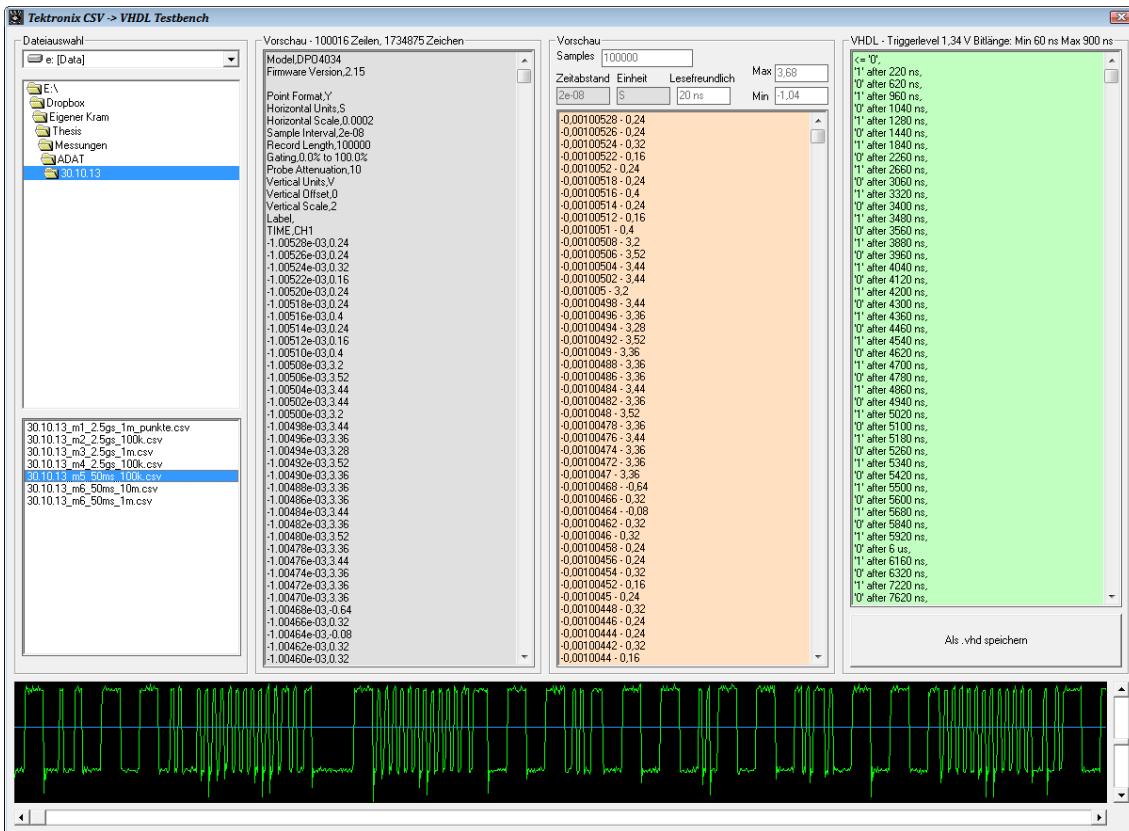


Abbildung 4.1: Das Programm „Tektronix CSV -&gt; VHDL Testbench“

beginnen.

Diese werden nun in einer For-Schleife mit `Record Length` Durchläufen eingelesen. Dazu wird zunächst vom Zeilenfang aus das nächste Komma gesucht, alles dazwischen wird als Zeitwert interpretiert. Der Rest (vom Komma bis zum Zeilenende) muss dementsprechend der gesuchte Messwert sein.

Die fertig geprästen Daten werden zur Kontrolle noch einmal in einem Textfeld (siehe Abbildung 4.1, hellroter Bereich) ausgegeben.

#### 4.2.9.2 Generieren des VHDL-Codes

Da es sich in diesem Fall um eine Digitalübertragung handelt, werden die Werte zunächst mit einer einfachen Triggerschwelle digitalisiert. Der Benutzer kann dabei den Trigger (blaue Linie) grafisch anhand des unten angezeigten Signalausschnitts einstellen<sup>2</sup>. Zusätzlich wird er nach erfolgter Konvertierung in VHDL-Code noch informiert, wie lang der kürzeste und der längste Abstand zwischen zwei Transitionen

<sup>2</sup> Scrollbalken rechts neben der Wellenform-Grafik.

ist, wodurch ein grob fehlerhaft eingestellter Trigger, welcher auch das Nachschwingen der Pegelwechsel erfasst, bereits bei rudimentärer Kenntnis über die Datenrate des Signals erkannt werden kann.

Die Messwerte werden nun in verwendbaren VHDL-Code umgewandelt, indem die Werte in eine passende Textform gebracht werden. Bei den Digitalwerten werden einfach nur die entsprechenden einfachen Anführungszeichen ergänzt, die Zeitwerte dagegen werden zunächst von einer Umwandlungsfunktion in die VHDL-Zeiteinheit (vgl. Tabelle 4.3) umgewandelt, bei der sie am leichtesten lesbar sind<sup>3</sup>. Außerdem werden sie als relative Werte zum Anfangszeitpunkt angegeben. Aus „-1.00508e -03“ wird demnach das wesentlich besser lesbare und VHDL-kompatible „220 ns“.

Um nicht unnötig Speicher zu verschwenden, werden aufeinanderfolgende Einträge, deren Digitalwert gleich ist, auf den jeweils ersten Wert (und damit auf den Wechsel) beschränkt.

Die nun fertigen Daten werden Zeile für Zeile in der Form

```

1  <= '0',
2  '1' after 220 ns,
3  '0' after 620 ns,
4  '1' after 960 ns,
5  '0' after 1040 ns,
6  '1' after 1280 ns,
```

Listing 4.13: Ausgabe von „Tektronix CSV -> VHDL Testbench“

in ein Textfeld (siehe Abbildung 4.1, grüner Bereich) ausgegeben. Bei Bedarf können sie jetzt entweder direkt per Copy & Paste in den entsprechenden Quellcode kopiert oder als VHDL-Datei gespeichert werden.

## 4.3 Constraints

Constraints sind Angaben, die einerseits dem Synthesizer Hinweise geben können, wie ein bestimmtes Code-Stück zu verstehen ist oder welchen Teil er in einer besonderen Weise optimieren soll, andererseits aber z.B. auch so elementare Operationen wie das Verknüpfen eines Signals (Netzes) des Quellcodes mit einem Pin am FPGA-Gehäuse ausführen.

### 4.3.1 Die .ucf-Datei

Zu jedem Projekt in ISE sollte immer auch eine \*.ucf-Datei gehören. UCF steht für „User Constraints File“.

---

<sup>3</sup>Für leichte Lesbarkeit sind hier zwei Faktoren ausgewählt worden: 1. keine Kommawerte; 2. Angabe immer mit dem Zeit-Suffix, der die kürzeste Zahl ergibt.

#### 4.3.1.1 Probleme mit mehreren .ucf-Dateien

Es ist möglich, mehrere .ucf-Dateien in einem Projekt zu verwenden. Die naheliegendste Option wäre es daher, verschiedene Konfigurationen für verschiedene FPGAs in verschiedene ucf-Dateien im selben Projekt zu schreiben.

Davor kann man allerdings nur warnen, da z.B. eine zweite .ucf-Datei mit anderem **CONFIG PART** (siehe unten) nicht wie vielleicht erwartet verwendet wird: Laut einer Präsentation von XILINX zum Thema UCF-Dateien ist bei Constraint-Konflikten immer die zuletzt gelesene Constraint aktiv: „If multiple constraints conflict, then the last constraint takes implicit priority over previous constraints“ [Xil10c, S. 6]

Das möglicherweise erwartete Verhalten – die .ucf-Dateien sind nur mit passendem **CONFIG PART** gültig – tritt hier nicht ein, da bei unpassendem **CONFIG PART** lediglich eine Fehlermeldung erscheint (siehe unten). Da der **CONFIG PART** von der zweiten .ucf-Datei noch überschrieben wird, bevor die Constraints ausgewertet werden (siehe oben), erscheint keinerlei Fehlermeldung, obwohl möglicherweise zahllose in der zweiten .ucf-Datei nicht vorhandene Constraints gesetzt sind.

Laut XILINX ist die Verwendung mehrerer .ucf-Dateien vor allem dafür gedacht, Timing-Constraints von „Placement-Constraints“ zu trennen oder extern generierte .ucf-Dateien komfortabel einbinden zu können:

- The ISE®tool allows multiple UCFs to be added to a project
- Convenient way to
  - Separate placement constraints from timing constraints
  - Add constraints provided by the tools (from the Architecture Wizard or the CORE Generator™ tool) without using copy & paste

[Xil10c, S. 6]

#### 4.3.1.2 Manuelles Eintragen der Constraints

Maximale Kontrolle über alle Constraints ist bei der direkten Bearbeitung der UCF-Datei gewährleistet. Die in diesem Projekt verwendeten Befehle sind:

```
1 CONFIG PART = xc3s200a-vq100-5;
```

Listing 4.14: Constraints: **CONFIG PART**

Mit dieser Zeile wird mittels **CONFIG** der zu verwendende FPGA (**PART**) eindeutig festgelegt. In den Projekteinstellungen von ISE ist der FPGA ebenfalls einstellbar.

Sollten sich die ISE-Einstellung und **CONFIG PART** unterscheiden (selbst das Umstellen des Speedgrades<sup>4</sup> zählt hier), so erhält der Benutzer beim Synthesieren eine Fehlermeldung. Dies hilft, wenn wie in diesem Fall, zwei UCF-Dateien für zwei verschiedene Ziel-FPGAs vorliegen, bei denen in der Regel auch die Pinbelegung anders sein wird.

```
1 CONFIG VCCAUX = 3.3;
```

Listing 4.15: Constraints: **CONFIG VCCAUX**

Hier wird mit dem **CONFIG**-Kommando dem Constraint **vccaux** der Wert 3,3 V zugewiesen (siehe 5.1.1.2 *Spannungsversorgung*).

```
1 TIMESPEC TS_takt = PERIOD "takt" 20 ns HIGH 50 %;
```

Listing 4.16: Constraints: **TIMESPEC** und **PERIOD**

Hier wird eine **TIMESPEC** definiert, die später mehreren Netzen zugewiesen werden kann. Mittels **PERIOD** wird hier ein Takt mit 20 ns Periodendauer und einem Taktverhältnis von 50% „High“ zu 50% „Low“ definiert.

→ Weitergehende Informationen zur **PERIOD**-Constraint sind unter anderem in „What Are PERIOD Constraints?“ [Xil07a] zu finden.

Wenn das Projekt später synthetisiert (siehe 4.4 *Synthese*) und implementiert (siehe 4.5 *Map und PAR*) wird, kann automatisch geprüft werden, ob diese Timing-Constraints eingehalten werden können bzw. um wie viel davon abgewichen wird.

Mit **NET** können einem „Netz“, also einer Leitung, mehrere Parameter zugewiesen werden:

```
1 NET "in_Platinentakt" TNM_NET = "takt";
```

Listing 4.17: Constraints: **NET**

Hier wird dem Eingangsnetz **"in\_Platinentakt"** über das oben definierte Netz **"takt"** die dazugehörige **TIMESPEC** zugewiesen.

```
1 NET "in_Platinentakt" LOC = P89 | IOSTANDARD = LVCMOS33;
```

Listing 4.18: Constraints: **NET**-Pinzuweisung

<sup>4</sup>Die Spartan-3A-FPGAs sind z.B. in Speedgrade 4 und 5 erhältlich. Speedgrade 4 ist dabei die „normale“ Ausführung, bei Speedgrade 5 sind schnellere Timings möglich.

Hier dagegen wird das Netz einem bestimmten Pin am Gehäuse des FPGAs zugewiesen. Außerdem wird der FPGA angewiesen, dass das Signal dem Standard LVCMOS33 entspricht, also ein 3,3 V-Logiksignal ist.

```
1 CONFIG PART = xc3s200a-vq100-5; # Wir benutzen Speedgrade 5
```

Listing 4.19: Constraints: Kommentare in .ucf-Dateien

Mittels `#Kommentar` lassen sich nützliche Zusatzinformationen als Kommentar hinzufügen.

#### 4.3.1.3 PlanAhead

In dem Programm PlanAhead, welches Teil der Entwicklungsumgebung ISE ist, lassen sich die Netze den entsprechenden FPGA-Pins einfach per Drag&Drop zuweisen. Auch die I/O-Standards können hier zugewiesen werden. Letztlich werden alle hier bearbeiteten Constraints aus der UCF-Datei gelesen und auch wieder dorthin zurückgeschrieben.

→ *Im Allgemeinen bleiben die von Hand eingetragenen Constraints erhalten bzw. werden von PlanAhead modifiziert, sofern der Benutzer dort eine entsprechende Änderung vorgenommen hat. In einem Fall während der Bearbeitung dieser Thesis ist es aber auch vorgekommen, dass ein Netz nachher zwei Pins zugewiesen war, von denen ein Pin am eingestellten FPGA überhaupt nicht existierte. Eine abschließende manuelle Kontrolle der UCF-Datei ist daher empfehlenswert.*

#### 4.3.1.4 Constraints-Editor

Als dritte Möglichkeit gibt es schließlich noch den „Constraints-Editor“ im „Tools“-Menü von ISE. Hier lassen sich die meisten Constraints ebenfalls mit einer grafischen Oberfläche bearbeiten.

→ *Auf Grund der hohen Kontrolle über die Constraints wurde im wesentlichen die manuelle Bearbeitung des reinen UCF-Textes bevorzugt. Lediglich das initiale Zuweisen der FPGA-Pins erfolgte mit PlanAhead. Die Verwendung der Schlüsselwörter TNM\_NET und TIMESPEC geht auf eine automatische Konvertierung beim testweisen Aufrufen des Constraints-Editors zurück.*

#### 4.3.2 Constraints im VHDL-Code

```
1 entity EinModul is
2   Port  ( clk : in std_logic;
3           In   : in bit;
```

```

4      Out : out std_logic := '0' );
5  attribute clock_signal : string;
6  attribute clock_signal of clk : signal is "yes";

```

Listing 4.20: Constraints im VHDL-Code

In Listing 4.20 ist zu sehen, wie Constraints auch lokal im VHDL-Code verwendet werden können. In diesem Fall soll mit der Constraint `clock_signal` extra noch einmal klar gemacht werden, dass es sich bei `clk` um ein Taktsignal handelt.

Dazu wird ein VHDL Attribut namens `clock_signal` zunächst definiert und in der folgenden Zeile dann der Constraint-Wert angegeben. Jede Constraint, die in VHDL verwendet werden soll, muss erst als VHDL-Attribut deklariert werden:

In VHDL code, constraints can be specified with VHDL attributes. Before it can be used, a constraint must be declared with the following syntax:

**attribute attribute\_name : string;** [Xil13b, S. 24]

Danach kann sie z.B. einem Signal oder einer Variable zugeordnet und mit einem Wert versehen werden.

## 4.4 Synthese

Bei der Synthese wird aus dem VHDL-Code eine Logik-Schaltung generiert. Außerdem könne hier bereits diverse Optimierungen wie z.B. das Entfernen unbenutzter Schaltungsteile oder das Minimieren von Logik-Ausdrücken vorgenommen werden.

### 4.4.1 Nicht synthetisierbare Konstrukte, die in der Simulation aber erlaubt sind

*Nicht alles was im Simulator funktioniert, ist auch synthesefähig:*

#### 4.4.1.1 Der Modulo-Operator

Der Modulo-Operator `mod` kann in der Simulation problemlos mit normalen Ganzzahlen verwendet werden. In der Synthese sind allerdings nur Zweierpotenzen für den Divisor erlaubt. Ist der Wertebereich des Dividenden hinreichend klein und wird nur auf einen bestimmten Rest geprüft, kann eventuell wie in Abschnitt 4.6.6.2 „*Bitstufing*“ eine einfache ODER-verknüpfte Abfrage der Zahl auf die einzelnen „Modulo = Rest“-Werte als Workaround verwendet werden.

#### 4.4.1.2 Triggern an an Flanken

Für den Simulator ISim ist es kein Problem, wenn der Code in einem Prozess sowohl von der steigenden als auch von der fallenden Flanke eines Signals getriggert wird, z.B. durch `if clk'event then`. Synthesierbar ist dies allerdings nicht, da die Flipflops im FPGA nur an einer der Flanken getriggert werden können.

#### 4.4.1.3 Wartebefehle

Der Befehl `after` kann nur in der Simulation verwendet werden (siehe 4.2.8 *Testbenches*). Ebenso verhält es sich mit `wait for`. Beide geben konkrete Wartezeiten vor, welche aber nicht synthetisierbar sind.

### 4.4.2 VHDL-Konstrukte, die bei der Synthese verloren gehen

Bei der Synthese wird die Sensitivitätsliste eines Prozesses in der Regel komplett verworfen und aus allen Signalen, auf die innerhalb des Prozesses zugegriffen wird, neu erstellt.

Wichtig ist die Sensitivitätsliste aber für die Simulation – ohne korrekte Sensitivitätsliste verhält sich die Simulation möglicherweise nicht wie das synthetisierte Projekt.

### 4.4.3 Synchron & Asynchron - Die Gefahr metastabiler Zustände

Da die Flipflops im FPGA Zeit brauchen, bis sie den angelegten Logikpegel übernommen haben, ist bei asynchronen Vorgängen ggf. eine Synchronisierung der Signale erforderlich, welche z.B. durch Einlesen der Signale in ein mit dem Zieltakt getaktetes Flipflop erfolgen kann.

Synchrones Abfragen asynchroner Signale ist allerdings ebenfalls möglich. Wenn dabei der Datentyp `std_logic` verwendet wird, werden mögliche Zwischenzustände des Eingangs ebenfalls abgebildet, was je nach Auswertung erwünscht sein kann oder auch nicht: Wenn z.B. explizit abgefragt wird, ob ein Signal „0“ ist, werden beim Ansteigen des Signals bei einem Pegelwechsel evtl. auch diverse Zwischenzustände wie z.B. ‚U‘ (Undefiniert) durchlaufen. Geht man nun einfach davon aus, dass das Signal entweder „0“ oder „1“ ist und schreibt den Code für „1“ in den Else-Zweig der obigen If-Abfrage, so wird dieser eben auch bei allen anderen Zuständen – unter anderem auch bei undefiniertem Pegel – aufgerufen. Bei Verwendung des Typs `bit` gehen diese Zustände verloren und es wird entweder Zustand „1“ oder „0“ angenommen, was bei der beschriebenen Abfrage zum erwarteten Ergebnis führt.

#### 4.4.4 Ressourcen-Optimierung

Wichtigster Punkt bei der Optimierung für einen FPGA ist es, immer „in Hardware zu denken“. Zweitwichtigster Punkt ist die Nutzung aller vorhandenen Ressourcen: Eine Funktion, die auch von einer noch nicht genutzten Spezialeinheit des FPGAs erledigt werden kann, sollte in der Regel nicht in normaler Logik realisiert werden, um den Verbrauch der Logikelemente (die sonst noch anderweitig genutzt werden könnten) nicht unnötig zu erhöhen.

Im Folgenden soll am Beispiel der NRZI-Dekodierung im ADAT-Dekodermodul gezeigt werden, wie die Programmierung nach klassischen Konzepten wie z.B. einem Array-Zugriff per Index in einem FPGA zu enormer (unnötiger) Ressourcenverwendung führen kann:

```

1 for i in 0 to AdatSchnittstellen-1 loop
2     --ADAT_Frame(i) (BitZaehler) := (Letztes_Bit(i) xor
3                                         IN_ADAT(i));
4     ADAT_Frame(i) := ADAT_Frame(i) (1 to 255) & (Letztes_Bit
5                                         (i) xor In_ADAT(i));--hier äquivalent, spart aber Slices
6 end loop;
7 Letztes_Bit := IN_ADAT; -- für nrzi merken

```

Listing 4.21: Ressourcenoptimierung am Beispiel der NRZI-Dekodierung

##### 4.4.4.1 Kurze Funktionsbeschreibung des Quelltextausschnitts

Dieser Codeausschnitt führt eine NRZI-M-Dekodierung aus und schreibt das Ergebnis Bit für Bit in einem Puffer. Sobald dieser voll ist wird das gesamte Array an den Ausgang gelegt (hier nicht zu sehen).

##### 4.4.4.2 Indexzugriff vs. Durchschieben der Bits

In der auskommentierten Zeile zwei ist der erste Ansatz zu sehen: In der Variable BitZaehler wird die aktuelle Bitnummer hochgezählt (hier nicht zu sehen). Dann wird mit dieser Bitnummer per Indexzugriff jeweils auf das passenden Array-Element ADAT\_Frame(i) (BitZaehler) zugegriffen. Bei der Synthese werden daraus einerseits natürlich eine große Menge von Flipflops für die einzelnen Bits, andererseits aber auch zwei große Multiplexer (1xEingang & 1xAusgang des Flipflops) für den Indexzugriff auf die einzelnen Bits.

Die Variante in Zeile drei wäre in der Programmierung für einen normalen Prozessor in der Regel deutlich ineffizienter, da jeweils alle bis auf das erste Bit um eine Position verschoben werden (die letzte Position nimmt das neu hinzukommende Bit ein).

In einem FPGA dagegen wird genau wie oben eine große Menge an Flipflops generiert. Auch der Multiplexer für die Ausgänge bleibt erhalten, da die Bits nach wie vor in einem Rutsch ausgelesen werden. Die Eingänge der Flipflops dagegen werden nun mit dem Ausgang des jeweils benachbarten Flipflops verbunden, wodurch ein langes Schieberegister entsteht. Die neuen Daten werden jetzt also nur noch über das eine Ende der Flipflop-Kette eingelesen.

Im direkten Vergleich ergab sich nach der Synthese eine Verringerung des Resourcenverbrauchs um 334 sogenannte „Slices“ (zum Vergleich: der hier verwendete FPGA XC3S200A hat insgesamt 1.792 Slices [Xil10b, S. 3]) bei damals zwei ADAT-Schnittstellen.

*Es sollte erwähnt werden, dass diese Optimierung hier nur aufgrund des speziellen Zugriffsmusters auf das Array möglich ist. Andere Zugriffsmuster müssen ggf. auf anderen Wegen optimiert werden.*

#### 4.4.4.3 Block-RAM

Der oben genannte Puffer wird in der Synthese als eine Reihe von Flipflops realisiert. Werden die Speicher noch größer, so empfiehlt sich das Nutzen der dedizierten Block-RAM-Einheiten des FPGA. Zwei Möglichkeiten zur konkreten Realisierung in VHDL werden im weiteren Verlauf der Thesis in den Abschnitten *Verwendung der Block-RAM Primitives* und *VHDL-Code so schreiben, dass Block-RAM erkannt wird* im Kontext des VHDL Moduls ADAT\_BLOCKRAM näher erläutert (siehe 4.6.7 *ADAT\_BLOCKRAM*).

### 4.5 Map und PAR (Place And Route)

#### 4.5.1 Map

Beim Map-Prozess wird die in der Synthese erzeugte Logik auf die Hardware eines speziellen FPGAs „gemappt“, die Logikelemente werden also an die vorhandene Hardware angepasst und dafür ggf. aufgeteilt.

#### 4.5.2 PAR

Bei PAR (Place And Route) werden die von Map angepassten Logikelemente in den einzelnen Zellen des FPGAs platziert (Place) und die Daten- und Taktsignale entsprechend geroutet. Taktsignale werden dabei bevorzugt auf die besonders gut angebundenen Global-Clock-Leitungen gelegt.

### 4.5.3 Ressourcen-Optimierung

Auch bei Map und PAR sind Optimierungen möglich. Einerseits können automatische Anpassungen wie z.B. eine Optimierung kombinatorischer Logik erfolgen, andererseits sind aber auch deutlich präzisere Eingriffe möglich.

#### 4.5.3.1 Manuelle Platzierung

Eine manuelle Platzierung der einzelnen Elemente wahlweise vor oder nach Place & Route ist im FPGA-Editor möglich, wurde allerdings aufgrund des hohen Zeitaufwandes hier nicht benutzt.

#### 4.5.3.2 Planung mit PlanAhead

Mit PlanAhead lässt sich nach Place & Route einerseits das sogenannte „Floorplan Design“ ansehen, andererseits ist es aber auch möglich, problematischen Elementen einen bestimmten Bereich im FPGA zuzuweisen, um sie z.B. näher an wichtiger Hardware zu platzieren. Im praktischen Test waren damit zwischenzeitlich deutliche Verbesserungen der kritischen Timings (ca. 10 %) möglich, sofern die Bereiche geeignet gewählt wurden.

## 4.6 Die VHDL-Module

Im Folgenden werden die im Rahmen dieser Thesis entwickelten VHDL-Module vorgestellt. Aus Platzgründen wird auf eine Erklärung der jeweiligen Testbenches verzichtet.

### 4.6.1 Übersicht

Alle Module bis auf das McBSP-Interface und die Taktgenerierung selber werden von ein und demselben Adat-Bit-Takt getaktet. Dieser kann entweder direkt von einem passenden Oszillator oder aus dem Taktgewinnungsmodul kommen.

Das McBSP-Interface dagegen wird vom DSP über die entsprechenden CLK-Leitungen getaktet.

Die Synchronisation zwischen McBSP-Takt und Adat-Bit-Takt sowie auch die Synchronisation aller anderen Einheiten erfolgt über ein einfaches Ereignissystem. Ist eine Einheit mit der Verarbeitung fertig, so ändert sie ein bestimmtes Bit an ihrem Ausgang. Dies erkennt das nachfolgende bzw. vorangehende Modul und leitet entsprechende Schritte ein: Der Puffer z.B. übernimmt neue Daten oder gibt den nächsten Datensatz aus.

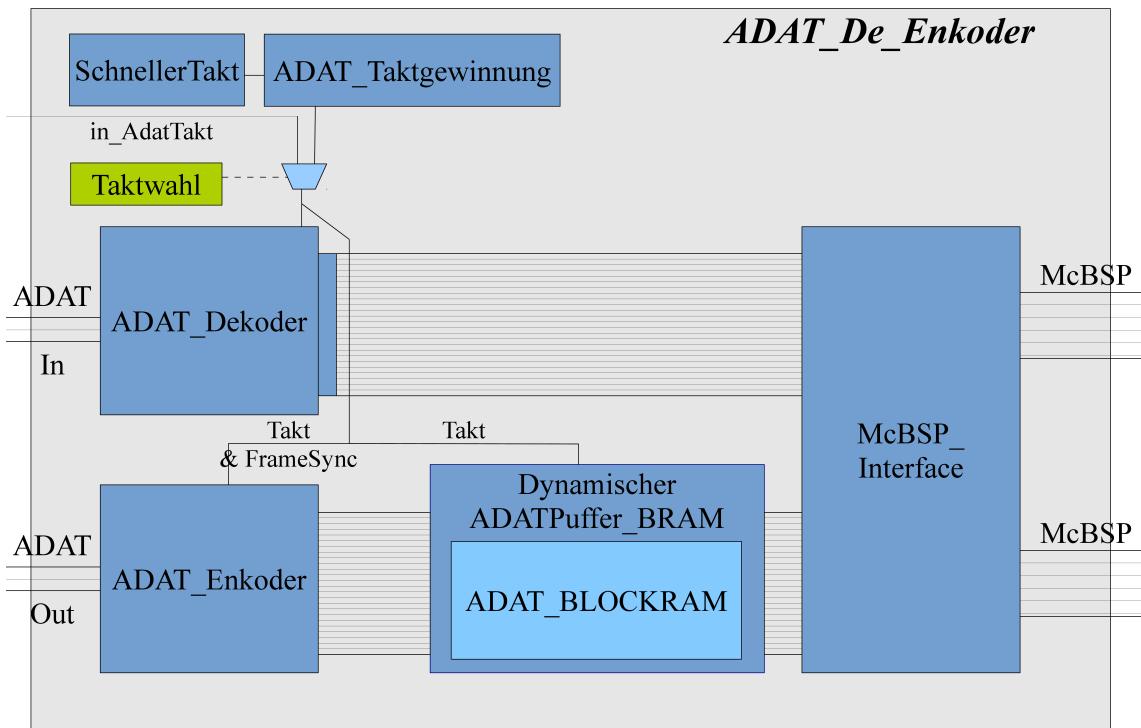


Abbildung 4.2: Übersicht über das Zusammenspiel der VHDL-Module

Das gesamte VHDL-Design skaliert dabei mit dem in **ADAT\_De\_Encoder** angegebenen **generic**-Parameter für die Anzahl der ADAT-Schnittstellen.

#### 4.6.2 Das Package ADAT

Datei: *ADAT.vhd*

Das VHDL-Package ADAT stellt häufig benutzte Konstanten, Datentypen und Funktionen zur Verfügung, die mit ADAT im Zusammenhang stehen.

##### 4.6.2.1 Konstante: **ADAT\_DATENMENGE**

```
1 constant ADAT_DATENMENGE : natural := (192+4);
```

Listing 4.22: Definition der Konstante **ADAT\_DATENMENGE**

Für die Größenangabe bei Arrays und die korrekte Indexberechnung in diesen ist diese Konstante gedacht, welche die Menge der Datenbits in einem ADAT-Frame angibt.

#### 4.6.2.2 Datentyp: ADAT\_Daten

```

1 type Audio8x24 is array (0 to 7) of bit_vector(23 downto 0)
      ;
2 type ADAT_Daten is record
3     Kanaele : Audio8x24;
4     UserBits : bit_vector(3 downto 0);
5 end record ADAT_Daten;

```

Listing 4.23: Definition des Datentyps ADAT\_Daten

Wichtigster Datentyp ist ADAT\_Daten. Dieser Datentyp stellt eine Struktur zur Speicherung aller Daten eines ADAT-Frames zur Verfügung. Der Zugriff auf das die Audiodaten von Kanal drei erfolgt z.B. mit Variablenname.Kanaele(3). Durch die Nutzung des Datentyps Audio8x24 ist aber z.B. auch das Kopieren aller Audio-kanäle gleichzeitig möglich (Audiodaten := Variablenname.Kanaele;).

#### 4.6.2.3 Datentyp: ADAT\_Daten\_Array

```

1 type ADAT_Daten_Array is array (natural range <>) of
      ADAT_Daten;

```

Listing 4.24: Definition des Datentyps ADAT\_Daten\_Array

Für die komfortable Speicherung der Daten mehrerer ADAT-Schnittstellen gibt es den Datentyp ADAT\_Daten\_Array. Dieser ist (wie der Name schon nahelegt) nichts weiter als ein Array von Elementen des Typs ADAT-Daten. Die Größe des Arrays muss während der Variablen-deklaration angegeben werden.

#### 4.6.2.4 Datentyp: ADAT\_Frame\_Array

```

1 type ADAT_Frame_Array is array (natural range <>) of
      bit_vector(0 to 255);

```

Listing 4.25: Definition des Datentyps ADAT\_Frame\_Array

Dieser Datentyp ist vor allem zur Speicherung mehrerer vollständiger ADAT-Frames gedacht. Im Gegensatz zu ADAT\_Daten\_Array werden hier auch die Bits gespeichert, die keine relevanten Daten enthalten (also die Bitstuffing- und Frame-Sync-Bits). Benutzt wird dies z.B. im ADAT-Dekoder, der erst alle Bits einliest, dann den gesamten Frame prüft (siehe 4.6.5.2 Fehlererkennung) und erst nach positivem Prüfungsergebnis die Daten aus dem Frame extrahiert.

#### 4.6.2.5 Funktion: AdatDatenArray\_TO\_Bitvector

```
1 function AdatDatenArray_TO_Bitvector (Datenarray : in
                                         ADAT_Daten_Array) return bit_vector;
```

Listing 4.26: Deklaration der Funktion AdatDatenArray\_TO\_Bitvector

Diese Funktion wandelt ein übergebenes ADAT\_Daten\_Array beliebiger Größe in ein entsprechend langes eindimensionales Array vom Typ bit\_vector um. Die Daten werden folgendermaßen angeordnet:

MSB		LSB	
ADAT_Daten_Array(0)		ADAT_Daten_Array(1)	...
Userbits   Kanal 7 ... Kanal 0	Userbits   Kanal 7 ... Kanal 0		...
Rückgabewert von AdatDatenArray_TO_Bitvector()			

Tabelle 4.4: Funktion AdatDatenArray\_TO\_Bitvector: Bitanordnung

#### 4.6.3 ADAT\_De\_Enkoder

Datei: ADAT-De-Enkoder.vhd

ADAT\_De\_Enkoder ist das sogenannte „Top-Modul“. Dieses ist vergleichbar mit der main()-Funktion in diversen Programmiersprachen wie z.B. C/C++. Von hier aus wird die gesamte Funktionalität gesteuert. Bei kleinen Projekten kann hier auch der gesamte VHDL-Code stehen. Bei größeren Projekten ist die Aufteilung in verschiedene VHDL-Module und die Instanziierung der entsprechenden Komponenten in der Regel unumgänglich. Im Gegensatz zur main()-Funktion kann man dem Code des Top-Moduls nicht ansehen, dass es sich um das Top-Modul handelt, da „Top-Modul“ einfach nur eine Eigenschaft in ISE ist, die immer nur genau einem VHDL-Modul innerhalb des Projektes zugewiesen werden kann.

Die grundlegende Struktur des Moduls wurde bereits in Abschnitt 4.6.1 „Übersicht“ erläutert.

Zusätzlich gibt das Modul auch noch die diversen LED-Signale weiter bzw. generiert diese.

→ Aufgrund der großen Zahl der Ein- und Ausgänge des Top-Moduls wird an dieser Stelle auf eine Darstellung der dazugehörigen Entity verzichtet. Im Wesentlichen handelt es sich dabei um die Daten- und Taktquellen der verschiedenen Untermodule.

#### 4.6.4 ADAT\_Taktgewinnung

Datei: *ADAT\_Taktgewinnung.vhd*

```

1 entity ADAT_Taktgewinnung is
2   generic (AdatSchnittstellen : integer := 3);
3   Port  ( clk           : in std_logic;
4             -- Daten-Eingang
5             In_ADAT        : in bit;
6             -- Takt-Ausgang
7             Out_ADAT_Clock : out std_logic := '0';
8             Frame_Sync     : out std_logic := '0' );
9 end ADAT_Taktgewinnung;
```

Listing 4.27: ADAT\_Taktgewinnung - Die Entity

Das Modul ADAT\_Taktgewinnung implementiert die in Abschnitt 2.3.2 (Taktgewinnung in einem reinen Digitalsystem) vorgestellte Methode zur Taktgewinnung aus dem ADAT-Signal.

Der Eingang `clk` nimmt den schnellen Systemtakt entgegen. `In_ADAT` ist der Eingang für das ADAT-Signal, zu dem der Takt ermittelt werden soll. `Out_ADAT_Clock` gibt den resultierenden ADAT-Bit-Takt aus, an `Frame_Sync` liegt das dazugehörige Frame-Sync-Signal an.

##### 4.6.4.1 Warum nicht drei Prozesse, sondern nur einer?

Auch wenn die Aufteilung der einzelnen Stufen (Frame-Sync-Erkennung, Frame-Längen-Messung und Bit-Takt-Generierung) in drei Prozesse naheliegend wäre, wurde darauf nach einigen Tests verzichtet. Das Problem bei drei Prozessen liegt darin, dass jeder der Prozesse auf die Ergebnisse des vorhergehenden Prozesses aufbaut. Auch wenn die Prozesse im FPGA vollkommen parallel laufen, sind sie doch getaktete Systeme.

Demnach kann ein Ergebnis, welches der erste Prozess zu beliebiger Zeit (aber mit der Taktflanke) ausgibt, im zweiten Prozess erst einen Takt später ankommen: Der zweite Prozess holt seine Eingangswerte zum selben Zeitpunkt ab, zu dem der erste Prozess sein Ergebnis ausgibt. Da das Signal aber erst einmal stabil am Eingang anliegen muss, bevor es gelesen wird, kann es erst bei der nächsten Taktflanke sinnvoll eingelesen werden: „The data must become valid at its input pins at least a setup time before the arrival of the active clock edge at its pin.“ [Xil12, S. 64] Folglich erhält man bei Verwendung von drei Prozessen eine signifikante (und unnötige) Verschiebung aller Ausgangssignale.

Wird nun aber nur ein Prozess verwendet, in dem die einzelnen Schritte zwar nacheinander erfolgen, die Ergebnisse der einzelnen Prozesse aber mit Variablen

(diese nehmen ihre Werte sofort an) weitergegeben werden, so beschränkt sich die Verzögerung auf den einen Takt, in dem der Prozess abgearbeitet werden muss.

#### 4.6.5 ADAT\_Dekoder

Datei: *ADAT\_Dekoder.vhd*

```

1 entity ADAT_Dekoder is
2     generic (AdatSchnittstellen : integer := 3);
3     Port (ADAT_Takt      : in std_logic;
4           ADAT_Sync      : in bit;
5           -- Daten-Eingänge
6           In_ADAT       : in bit_vector(0 to
7                           AdatSchnittstellen-1);
8           -- Daten-Ausgang
9           Out_Daten      : out ADAT_Daten_Array(0 to
10                      AdatSchnittstellen-1);
11           Out_FrameOK   : out bit_vector(0 to
12                      AdatSchnittstellen-1);
13           Out_NeueDaten : out bit      );
14 end ADAT_Dekoder;
```

Listing 4.28: ADAT\_Dekoder - Die Entity

An den Eingängen ADAT\_Takt und ADAT\_Sync wird der von ADAT\_Taktgewinnung generierte Takt inklusive Frame-Sync entgegengenommen. Der Prozess Dekodierung wird immer zur steigenden Flanke von ADAT\_Takt ausgeführt:

##### 4.6.5.1 Einlesen und NRZI-Dekodierung

Zunächst werden die ADAT-Frame-Daten an den ADAT-Eingängen In\_ADAT in einen Puffer vom Typ ADAT\_Frame\_Array gespeichert. Die NRZI-Dekodierung erfolgt dabei direkt beim Einlesen:

```

1 for i in 0 to AdatSchnittstellen-1 loop
2     ADAT_Frame(i) := ADAT_Frame(i)(1 to 255)
3             & (Letztes_Bit(i) xor In_ADAT(i));
4 end loop;
5 Letztes_Bit := IN_ADAT; -- für nrzi merken
```

Die NRZI-Dekodierung liegt hier in dem Codeabschnitt Letztes\_Bit(i) **xor** In\_ADAT(i). Der Wert der Eingänge In\_ADAT wird jeweils mit dem Wert, den sie im vergangenen Taktzyklus hatten, mit einem Exklusiv-Oder verknüpft. Waren sie

unterschiedlich ergibt sich eine „1“, waren sie gleich ist es eine „0“, was genau der Dekodierung von NRZI-M entspricht.

Während dem Einlesen läuft ständig ein Bitzähler mit, der die Anzahl der empfangenen Bits pro Schnittstelle angibt. Sind alle 256 Bits einer ADAT-Frames eingelesen worden, so wird der Zähler zurückgesetzt. Die eigentlichen Daten werden aus dem Puffer extrahiert (einfaches Zuweisen der Werte von Ausschnitten des Puffer-Arrays an die entsprechenden Elemente der Datenstruktur des Ausgangssignals) und an den Ausgang des Dekoders gelegt.

#### 4.6.5.2 Fehlererkennung

Bevor die nachfolgenden Module mit einer Änderung am Ausgang Out\_NeueDaten informiert werden, dass ein weiterer Frame empfangen wurde und abgeholt werden kann, erfolgt noch ein Prüfung des Frames auf Fehler.

Da ADAT eigentlich keinerlei Fehlererkennungsmechanismen wie z.B. Checksummen oder Paritätsbits unterstützt, können lediglich die konstanten Bits (also Bitstuffing und Frame-Sync) auf ihre korrekten Werte überprüft werden. In der Simulation des Dekoders zeigte sich allerdings bereits, dass auch dies schon ein guter Indikator für fehlerhaft empfangene Frames ist, wie sie z.B. durch einen leicht verschobenen Takt entstehen könnten.

#### 4.6.5.3 Frame-Sync

Tritt eine Frame-Sync-Puls auf, so wird einfach der Bitzähler auf Null zurückgesetzt, ein eventuell bereits teilweise empfangener ADAT-Frame wird also einfach verworfen bzw. überschrieben.

#### 4.6.6 ADAT\_Enkoder

Datei: ADAT\_Enkoder.vhd

```

1 entity ADAT_Enkoder is
2   generic( AdatSchnittstellen : integer := 3);
3   Port( ADAT_Frame_Sync: in bit;
4         ADAT_Bit_Clock: in std_logic;
5         Daten: in ADAT_Daten_Array(0 to
6                               AdatSchnittstellen-1);
7         DatenUebernommen : out bit;
8         Out_ADAT : out bit_vector(0 to
9                               AdatSchnittstellen-1) := (others => '0');
10        );
11 end ADAT_Enkoder;
```

Listing 4.29: ADAT\_Enkoder - Die Entity

Genau wie beim Dekoder werden an den Eingängen ADAT\_Takt und ADAT\_Sync der von ADAT\_Taktgewinnung generierte Takt und Frame-Sync entgegengenommen. Daten ist der Eingang des Enkoders mit den rohen ADAT-Daten. An Out\_ADAT werden die resultierenden ADAT-Signale ausgegeben.

#### 4.6.6.1 Steuerung per Statemachine

Da die einzelnen Phasen beim Senden sich signifikant unterscheiden, werden diese mittels einer Statemachine gesteuert: In der Phase ADAT\_Userbits, mit der ein Frame immer beginnt, werden die Userbits gesendet. Diese liegen im Datentyp ADAT\_Daten als einfacher `bit_vector` vor, es muss also lediglich die Bitnummer mitgezählt werden und das entsprechende Bit (von jeder ADAT-Schnittstelle) ausgegeben werden. Die bei der Ausgabe immer (in jeder Phase) erforderliche NRZI-M-Kodierung erfolgt einfach, in dem der ADAT-Ausgang invertiert wird, sofern eine „1“ in den Daten vorliegt (bei einer „0“ bleibt der Pegel gleich). Nachdem alle Userbits gesendet wurden, wird in die Phase ADAT\_AudioDaten gewechselt.

In der Phase ADAT\_AudioDaten müssen zusätzlich noch die einzelnen Kanäle durchgegangen werden, ein Kanalzähler setzt folglich den hier benötigten lokalen Bitzähler entsprechend zurück. Sobald die Audiodaten komplett gesendet wurden, wird mit einer Änderung an Daten übernommen der nächste Datensatz angefordert<sup>5</sup> und in die Phase ADAT\_Sync gewechselt.

In der Phase ADAT\_Sync wird das Frame-Sync-Signal erzeugt, in dem nichts getan wird. Sobald der Phasenübergreifende Bitzähler 255 erreicht, wird wieder in die Phase ADAT\_Userbits gewechselt.

#### 4.6.6.2 Bitstuffing

Das Bitstuffing erfolgt über den Phasenübergreifenden Bit-Zähler, der die jeweils aktuelle Bitnummer im ADAT-Frame angibt. Bei jedem fünften Bit wird so anstatt eines Datenbits eine „1“ (also ein Pegelwechsel in NRZI-M) ausgegeben.

Da die Modulo-Operation `mod` nur mit 2-er-Potenzen synthetisierbar ist (siehe 4.4.1 *Nicht synthetisierbare Konstrukte*), wurde hier eine lange Oder-Verknüpfung von Vergleichen der Bitnummer mit den zu erwartenden, ohne Rest durch fünf teilbaren Zahlen verwendet.

<sup>5</sup> Warum werden die Daten schon hier angefordert? Einerseits ist dies der frühestmögliche Zeitpunkt zum Anfordern der Daten (vorher werden die aktuell anliegenden Daten noch benötigt), andererseits müssen die Daten ankommen, bevor der neue Frame anfängt, was aufgrund des getakteten Systems auch Zeit benötigt (siehe 4.6.4.1 *Warum nicht drei Prozesse, sondern nur einer?*).

### 4.6.7 ADAT\_BLOCKRAM

Datei: ADAT\_BLOCKRAM.vhd

```

1 entity ADAT_BLOCKRAM is
2     generic( ADAT_SCHNITTSTELLEN : natural := 3;
3                 PUFFER_LAENGE          : natural := 6 );
4     port(clk   : in  std_logic;
5             we    : in  std_logic;
6             en    : in  std_logic;
7             addr  : in  natural;
8             di    : in  bit_vector(ADAT_DATENMENGE*
9                               ADAT_SCHNITTSTELLEN-1 downto 0);
9             do    : out bit_vector(ADAT_DATENMENGE*
10                           ADAT_SCHNITTSTELLEN-1 downto 0) );
10 end ADAT_BLOCKRAM;

```

Listing 4.30: ADAT\_BLOCKRAM - Die Entity

Das VHDL-Modul ADAT\_BLOCKRAM stellt einen in der Datenbreite konfigurierbaren RAM dar, der beim Synthesisieren als großer RAM erkannt und daher als Block-RAM implementiert wird. Zentrales Problem dabei ist einerseits die Erkennung als RAM, andererseits die Datenbreite: Die maximal nutzbare Datenbreite eines Block-RAM-Moduls beträgt nur 32 Bit. [Xil05, S.10, Tabelle 6]

Bei der Benutzung von Speicherelementen gibt es verschiedene Möglichkeiten, dem Synthesizer klar zu machen, dass er Block-RAM verwenden soll:

#### 4.6.7.1 Verwendung der Block-RAM Primitives

Alle denkbaren Block-RAM-Konfigurationen einer Block-RAM-Einheit sind als sogenannte „Primitives“ verfügbar und direkt als Instanz der Komponenten der Bibliothek UNISIM verwendbar. Diese Instanzen lassen sich auch zu größeren Speichern kaskadieren, sind aber nicht so einfach für beliebige Datenbreiten verwendbar, weshalb auf die Verwendung der „Primitives“ hier verzichtet wurde.

→ Code-Beispiele und weitere Informationen für die Verwendung der einzelnen Block-RAM-Elemente finden sich in [Xil13a] auf den Seiten 199 bis 463.

#### 4.6.7.2 VHDL-Code so schreiben, dass Block-RAM erkannt wird

Eine weitere Möglichkeit ist es, seinen Speicher so zu schreiben, dass der Synthesizer diesen als im Block-RAM umsetzbaren Speicher erkennt und dementsprechend als Block-RAM synthetisiert.

In [Xil13c] werden ab Seite 133 diverse Codebeispiele in VHDL (und Verilog) gegeben, die von XST – dem Synthesewerkzeug in der Entwicklungsumgebung ISE – automatisch als Block-RAM erkannt werden.

Die dort vorgestellten RAMs gibt es in drei Varianten:

- **Write-First**

Der RAM gibt bei aktivem Enable-Eingang mit jedem Takt entweder die Daten aus, die an der angegebenen Adresse liegen (Write-Enable inaktiv), oder er liest die am Eingang liegenden Daten an die angelegte Adresse im RAM ein und gibt zusätzlich den gerade gelesenen Wert am Ausgang wieder aus.

- **Read-First**

Der RAM gibt bei aktivem Enable-Eingang mit jedem Takt die Daten aus, die an der angegebenen Adresse liegen. Ist zusätzlich Write-Enable aktiv so werden vorher noch die Daten vom Eingang in den RAM geschrieben (ebenfalls an die angelegte Adresse).

- **No-Change**

Ein RAM im „No-Change“-Modus kann nie gleichzeitig lesen und schreiben. Bei aktivem Enable-Eingang verhält es sich folgendermaßen: Ist Write-Enable aktiv, werden die Daten vom Eingang an die angelegte Adresse gespeichert. Ist Write-Enable inaktiv, werden die Daten der angelegten Adresse ausgegeben.

In diesem Fall wurde das Beispiel „Single-Port RAM In No-Change Mode“ [Xil13c, S. 142-143] als Vorlage verwendet. Die Wahl fiel auf No-Change, da das ganze als Ringpuffer verwendet werden soll (siehe unten), bei dem Ein- und Ausgabe der Daten zu unterschiedlichen Zeitpunkten erfolgen.

In Listing 4.30 ist die Entity des resultierenden VHDL-Moduls zu sehen. Einzige Änderung gegenüber dem Quelltext der Vorlage ist die Einführung von zwei generischen Parametern (`ADAT_SCHNITTSTELLEN` und `PUFFER_LAENGE`), mit denen Breite und Länge des Block-RAMs eingestellt werden kann, wobei die Breite als Anzahl der verwendeten ADAT-Schnittstellen angegeben wird.

#### **4.6.8 DynamischerADATPuffer\_BRAM**

*Datei: DynamischerADATPuffer\_BRAM.vhd*

```

1 entity DynamischerADATPuffer_BRAM is
2   generic (PUFFERLAENGE : integer := 2;
3             PUFFERBREITE : integer := 3 );
4   Port (Reset : in bit;
```

```

5      clk : in std_logic;
6      Eingang_NeueDatenAngekommen : in bit;
7      Eingang : in ADAT_Daten_Array(0 to
8                                PUFFERBREITE-1);
9      Ausgang_NaechsterDatensatz : in bit;
10     PufferVoll : out bit := '0';
11     PufferLeer : out bit := '1';
12     PufferUeberlauf : out bit := '0';
13     PufferUnterlauf : out bit := '0';
14     Ausgang : out ADAT_Daten_Array(0 to
15                                PUFFERBREITE-1) );
16 end DynamischerADATPuffer_BRAM;

```

Listing 4.31: DynamischerADATPuffer\_BRAM - Die Entity

Über die Generic-Parameter lassen sich Pufferlänge und -breite einstellen. Über eine Änderung des Bits `Eingang_NeueDatenAngekommen` wird dem Puffer mitgeteilt, dass neue Daten an seinem Eingang anliegen, die er nun übernehmen kann. `Ausgang_NaechsterDatensatz` fordert auf die gleiche Weise neue Daten am Ausgang des Puffers an.

#### 4.6.8.1 Funktionsweise

*Die hier erwähnte „zuerst schreiben oder zuerst lesen?“-Problematik lässt sich trotz der Namensähnlichkeit nicht mit den in 4.6.7.2 erwähnten RAM-Typen lösen und sollte nicht mit dem dort verwendeten Mechanismus verwechselt werden.*

DynamischerADATPuffer\_BRAM kapselt das oben beschriebene ADAT\_BLOCKRAM -Modul und führt die entsprechenden Adressberechnungen zur Verwendung des Block-RAMs als Ringpuffer durch. Außerdem sorgt es dafür, dass bei gleichzeitig angeforderten Schreib und Lese-Operationen (diese sind möglich, da die Module an Ein- und Ausgang des Puffers nicht direkt miteinander kommunizieren) beide nacheinander ausgeführt werden. Ob dabei der Lese- oder der Schreibzugriff Priorität hat, hängt davon ab, mit welchem Pufferproblem am ehesten gerechnet wird: Droht der Puffer eher überzulaufen, würde es sich lohnen den Lese-Befehl zuerst auszuführen, beim Unterlauf wäre es entsprechend der Schreibbefehl. Um die Latenz zwischen ADAT-Ein- und Ausgang möglichst gering zu halten, wird in diesem Fall die Schreiboperation zuerst ausgeführt<sup>6</sup>.

<sup>6</sup>Angenommen, der ADAT\_Encoder hat gerade seinen Frame vollständig gesendet und fordert nun den nächsten Datensatz vom Puffer an. Ist der Puffer nun aber leer, so wird zunächst der selbe Frame noch einmal gesendet, da der Puffer im „zuerst-Lesen-Modus“ keine neuen Daten an den Ausgang legen kann. Der zu schreibende Frame wird zwar danach in den Puffer aufgenommen, kommt beim ADAT-Encoder aber erst später an, da bei Pufferproblemen

Bei Pufferproblemen (Über- bzw. Unterlauf) wird die problematische Lese- bzw. Schreiboperation verworfen, um dem Puffer die Rückkehr in einen normalen Status zu ermöglichen.

#### 4.6.8.2 Adressberechnung

Das Verhalten des Block-RAMs als Ringpuffer wird durch entsprechende Adressberechnungen erzielt: In der Variablen `Anfang` wird jeweils die aktuelle Startadresse gespeichert, an der der jeweils älteste Datensatz zu finden ist. Über die Variable `Laenge` wird der aktuelle Füllstand des Puffers verwaltet. Gleichzeitig ist über die Kombination der beiden Variablen auch die Adresse berechenbar, an der das nächste Pufferelement abgelegt werden soll:

$$\text{BR\_ADDR}_{\text{Neu}} = (\text{Anfang} + \text{Laenge}) \bmod \text{PUFFERLAENGE}$$

Durch die Modulo-Operation an dieser Stelle sind für `PUFFERLAENGE` nur noch Zweierpotenzen erlaubt (siehe 4.4.1.1 *Der Modulo-Operator*).

#### 4.6.8.3 Pufferstatus

Die Ausgänge `PufferVoll`, `PufferLeer`, `PufferUeberlauf` und `PufferUnterlauf` geben den aktuellen Status des Puffers aus. Diese Bits werden später im Top-Modul (siehe 4.6.3 *ADAT\_De\_Enkoder*) direkt an die entsprechenden LEDs auf der Platine weitergeleitet. Das Signal `PufferNormal` fehlt hier noch – es wird im Top-Modul durch eine direkte Logikverknüpfung aus den anderen Signalen hergeleitet.

`PufferLeer` und `PufferVoll` werden durch einen Vergleich von `Laenge` mit „0“ bzw. `PUFFERLAENGE` nach den RAM-Operationen ermittelt. Die Signale für Unter- bzw. Überlauf werden „1“, wenn dieser Vergleich auch vor einer auszuführenden Lese- bzw. Schreiboperation schon positiv ausfällt.

#### 4.6.8.4 Wartezeiten

Da bei allen RAM-Operationen zunächst die Adresse und der Wert von Write-Enable an den RAM angelegt werden müssen, ergibt sich naturgemäß eine Verzögerung von einem Takt, bis die entsprechende Operation ausgeführt wird. Da der Block-RAM zusätzlich aber auch noch eine steigende Flanke an seinem Takteinangang benötigt, damit er die Operation auch wirklich ausführt, ergibt sich hierdurch eine Verzögerung um einen weiteren Takt. Schließlich müssen Adresse und

---

die jeweils problematischen Lese- bzw. Schreiboperationen verworfen werden (Hier wird also die Leseoperation verworfen). Wird stattdessen allerdings zuerst geschrieben, so ist der Puffer bereits gefüllt, wenn die (gleichzeitig angeforderte) Leseoperation erfolgt.

Write-Enable erst einmal stabil am Eingang des RAMs anliegen, bevor die Auswertung erfolgen darf. Zu diesem Zweck wird nach jeder RAM-Operation das Bit VerzoegerterRamTakt\_noetig gesetzt. Ist dieses gesetzt, so werden keinerlei RAM-Operation durchgeführt (diese werden aber auch nicht verworfen, sondern lediglich für den nächsten Takt „aufgehoben“) und das Bit sofort wieder gelöscht.

#### 4.6.9 McBSP\_Interface

Datei: *McBSP-Dekoder\_1Puffer\_Parallel.vhd*

```

1 entity McBSP_Interface is
2     generic ( SCHNITTSTELLEN : natural := 3;
3             FSR_Aktiv : bit := '1';
4             CLKR_Aktiv : bit := '0';
5             FSR_Verzögerung : natural range 1 to 2 := 1;
6             FSX_Aktiv : bit := '1';
7             CLKX_Aktiv : bit := '1';
8             FSX_Verzögerung : natural range 1 to 2 := 1 );
9     PORT(CLKR : IN bit;     CLKX : IN bit;
10        FSR : IN bit;      FSX : OUT bit := '0';
11        DR : IN bit;       DX : OUT bit := '0';
12        Daten_Eingang : in ADAT_Daten_Array(0 to
13                                         SCHNITTSTELLEN-1);
14        Daten_Ausgang : out ADAT_Daten_Array(0 to
15                                         SCHNITTSTELLEN-1);
16        Daten_komplett_empfangen : OUT bit := '0';
17        Neue_Daten_Zum_Senden : in bit; );
18 end McBSP_Interface;
```

Listing 4.32: McBSP\_Interface - Die Entity

Wie auch der McBSP in den DSPs der C6000-Reihe ist auch hier das McBSP-Interface an vielen Stellen konfigurierbar: SCHNITTSTELLEN gibt wie immer die Anzahl der ADAT-Schnittstellen an und bestimmt hier die Framelänge. Mit den diversen \_Aktiv-Variablen wird die Polarität der Signale FSR, CLKR, FSX und CLKX eingestellt. FSR\_Verzögerung und FSX\_Verzögerung geben die Verzögerung in Bits gemäß [Tex06a, S.28, Fig. 13] an (siehe auch Abbildung 3.3 in Abschnitt 3.4 „Ansteuerung im *FPGA*“).

→ Die gesamte McBSP-Dekodierung und -Kodierung erfolgt nach dem bereits in Abschnitt 3.4 „Ansteuerung im *FPGA*“ beschriebenen Prinzip.

# 5 Realisierung der Hardware

## 5.1 Die Schaltung

### 5.1.1 Der FPGA

#### 5.1.1.1 Über Bänke, DCMs und globale Taktnetze

Die Pinbelegung der Datenpins bei dem verwendeten FPGA ist praktisch beliebig anpassbar. Durch sinnvolles Anordnen der Pins gemäß der FPGA-Architektur ist allerdings ein deutlich effizienterer Betrieb möglich.

**Bänke** Zunächst einmal ist festzuhalten, dass beim hier verwendeten FPGA die Pins in vier sogenannte „Bänke“ aufgeteilt sind. Jede Bank hat ihre eigene Spannungsversorgung (siehe 5.1.1.2 *Spannungsversorgung*) über die auch die jeweiligen Ein- und Ausgangspegel festgelegt werden.

Die Pins einer Bank sind hier jeweils an einer Seite des FPGAs gruppiert<sup>1</sup>. [Xil10b, S. 74]

**DCMs** Die vier DCM-Module des FPGAs ermöglichen die Ableitung von benutzerdefinierten Takt aus beliebigen Eingangstakten. Sie sind jeweils in den vier Quadranten des FPGAs angeordnet und haben eine besonders gute Anbindung an die globalen Taktnetze.

**Globale Taktnetze** Da die Taktleitungen besondere Anforderungen an Verzögerung und Signalintegrität stellen, gibt es die sogenannten „Global Clocks“. Diese Leitungen sind mit dem ganzen FPGA verbunden und ermöglichen eine besonders effektive Taktverteilung.

Daneben gibt es noch die nicht ganz so gut angebundenen „Left Half Clock“ und „Right Half Clock“-Leitungen, die nur eine FPGA-Hälfte erreichen.

**Effektives Takt-Routing** Aufgrund der internen Architektur des FPGAs ist es sinnvoll, zusammengehörige Taktsignale auf gemeinsam verlaufenden Taktleitungen zu Routen (siehe Abbildung E.13 im Anhang). Dazu müssen die Taktsignale an

<sup>1</sup>Bank 2 verläuft hier leicht um die beiden unteren Ecken, hat also ein paar Pins mehr.

den entsprechenden zusammengehörigen Pins liegen. Auch bei der Verwendung von DCMs sollte für optimale Ergebnisse die Anbindung der Taktleitungen beachtet werden.

### 5.1.1.2 Spannungsversorgung

Die XILINX-Spartan-3A Serie benötigt eine Reihe verschiedener Spannungen:

**VCCINT** Der FPGA-Kern wird mit 1,2 V versorgt. Gemäß [Xil10b, S.15, Tabelle 10] liegt der Ruhestrom für VCCINT für die normale „commercial“-Ausführung des FPGAs bei maximal 50 mA. Die maximale Stromaufnahme liegt laut TEXAS INSTRUMENTS (welche unter anderem Referenzdesigns für die Spannungsversorgung der Spartan 3A-Serie zur Verfügung stellen) bei maximal 350 mA  $\pm 5\%$  [Tex13]

Der Low-Drop-Spannungsregler LD1117 liefert typ. 950mA, mindestens aber 800mA. Damit ist er für die Versorgung des FPGAs mehr als ausreichend überdimensioniert und wird nicht an seiner Leistungsgrenze betrieben, was vor allem auf Grund der Wärmeentwicklung wichtig ist (siehe 5.2.5 *Kühlung des Spannungsreglers*).

**VCCO 1–4** Die VCCO-Pins versorgen die einzelnen FPGA-Bänke (siehe 5.1.1.1 *Über Bänke, DCMs und globale Taktnetze*) und legen deren Eingangs-Schwellwertspannungen [Xil10b, 66, Tabelle 57] sowie den Ausgangsspannungsbereich [Xil11, S. 353] für einige I/O-Standards fest . Da sowohl der DSP als auch die ADAT-Buchsen in diesem Fall mit 3,3 V arbeiten, wird VCCO in dieser Schaltung ebenfalls mit 3,3 V verbunden.

Auch nicht benutzte Bänke müssen angeschlossen werden<sup>2</sup>.

**VCCAUX** VCCAUX stellt eine Hilfsspannung zur Verfügung („auxiliary power supply pin“[Xil10b, S. 66]). Diese kann bei der Spartan-3A-Serie wahlweise auf 2,5 V oder auf 3,3 V gelegt werden, um den Spannungsregler für den anderweitig nicht benötigte Spannungswert einsparen zu können. Neben den DCMs (siehe 5.1.1.1) und den Konfigurationspins versorgt VCCAUX auch die JTAG-Pins.

Da für diese Schaltung lediglich 3,3 V benötigt werden, wird VCCAUX hier ebenfalls an 3,3 V angeschlossen. Ein 2.5 V Spannungsregler ist somit nicht mehr nötig.

→ *Der gewählte Spannungswert muss als Constraint im ISE-Projekt angegeben werden (siehe 4.3 Constraints).*

**Sequentielles Einschalten der Spannungsquellen?** Ein sequentielles Einschalten der Spannungsquellen ist bei der Spartan-3A-Serie nicht nötig, da die POR

<sup>2</sup>,„All must be connected.“[Xil10b, S.66, Tabelle 57]

(Power On Reset)-Schaltung des FPGA diesen erst startet, wenn VCCINT, VC-CAUX und VCCO<sup>3</sup> ihre korrekten Spannungspegel erreicht haben. [Xil11, S. 470]

Beim Spartan-3A ist sogar uneingeschränkt „Hot Swap“ erlaubt, d.h. das Einsticken der nicht laufenden FPGA-Platine in eine laufendes System. Eine bestimmte Reihenfolge beim Einschalten der Spannungsversorgungen (VCCINT nach VC-CAUX) verhindert allerdings unnötig hohe Ströme im Einschaltmoment. [Xil11, S. 471]

### 5.1.1.3 Taktquellen

Als „schnelle“ Taktquelle dient ein 50 MHz-Oszillator in Standardbeschaltung nach Datenblatt. Aus dem 50 MHz-Takt wird im FPGA über das DCM-Modul (eine einstellbare PLL) dann der eigentliche „schnelle Takt“ generiert. Ein zweiter Oszillator mit 12,288 MHz stellt einen internen ADAT-Takt zur Verfügung.

→ Die Oszillatorkreisfrequenz von 12,288 MHz gilt nur für eine Audio-Abtastrate von 48 kHz. Falls andere Abtastraten eingesetzt werden sollen, muss die Frequenz entsprechend angepasst werden:  $f_{\text{Oszillator}} = f_{\text{Abtast}} \cdot 256$

Der Takt für das McBSP-Modul kommt direkt vom DSP.

### 5.1.1.4 Konfiguration

Der hier verwendete Spartan-3A-FPGA besitzt keinen eigenen Konfigurationsspeicher, er muss die Konfiguration (so zu sagen sein „Programm“) also von außen erhalten.

Von den zahlreichen Möglichkeiten, die im *Spartan-3 Generation Configuration User Guide* [Xil09] beschrieben werden, wurden hier die zwei Varianten aus der Application Note *XAPP986 - Bulletproof Configuration Guide for Spartan-3A FPGAs* [Xil07b] gewählt:

**JTAG** JTAG wird hier vor allem für Entwicklungs- und Debug-Zwecke zur Verfügung gestellt. Außerdem wird der hier ausgewählte PROM-Chip ebenfalls über die JTAG-Schnittstelle beschrieben, bevor er als nicht-flüchtiger Konfigurationsspeicher seinen Zweck erfüllen kann.

**Master Serial Mode** Im „Master Serial Mode“ mit einem PROM der XILINX-XCF-Serie wird der PROM-Chip vom FPGA mit dem Takt aus dessen internem Oszillator versorgt, benötigt also keine zusätzliche Taktquelle. Der PROM-Chip

<sup>3</sup>Spartan-3A/3E: VCCO\_2; Spartan-3: VCCO\_4

wiederum hält den FPGA über den Pin INIT\_B solange im Reset, bis seine eigene Spannungsversorgung stabil ist [Xil07b, S.3] und er die Konfigurationsdaten ausliefern kann.

Sobald sowohl FPGA als auch PROM bereit sind, werden die Konfigurationsdaten übertragen. Sobald der FPGA an seinem DONE-Pin anzeigt, dass er fertig konfiguriert ist („High“-Pegel), deaktiviert er damit den PROM [Xil07b, S.3], der im Standby nun nur noch sehr wenig Strom verbraucht [Xil10a, S. 15].

## 5.1.2 ADAT-Schnittstellen

Wie bereits in (siehe 2.1.3 *ADAT-Lightpipe*) erwähnt, verwendet ADAT die Buchsen und Kabel der optischen S/PDIF-Schnittstelle.

### 5.1.2.1 Wandlung der optischen Signale

Im Datenblatt des ADAT-Dekoders AL1402G von WAVEFRONT wird die Kombination von TORX173/TOTX173 als ADAT-Buchsen im Referenzschaltplan vorgeschlagen [Wav05b]. Beide Buchsen sind auf 5 V-Logikpegel ausgelegt.

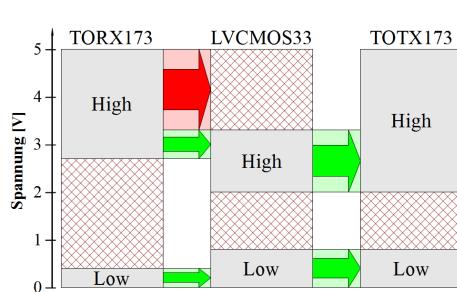


Abbildung 5.1:  
Kompatibilität der Logik-Pegel der Sender/Empfänger TOTX/TORX-173 mit dem I/O-Standard LVCMOS33 der Spartan-3A-Serie.  
Alle Pegel nach [Tos01a, S. 2][Xil10b, S. 16][Tos01b, S. 2] und den jeweiligen Versorgungsspannungen

Die Empfangsbuche TORX173 von TOSHIBA ist vom Low-Pegel her zwar kompatibel zum I/O-Standard LVCMOS33 der Spartan-3A-Serie, für High Pegel ist aber auf beiden Seiten kein direkter Maximalwert angegeben. Geht man davon aus, dass bei beiden der gültige Pegelbereich bis zur Versorgungsspannung reicht, so geht der TORX173 weit über die 3,3 V hinaus, die beim FPGA folglich als maximaler „High“-Pegel zulässig wären. Selbst die „Absolute Maximum Ratings“ von 4,6 V [Xil10b, S. 11] an den Eingängen des FPGAs wären mit den daraus folgenden 5 V überschritten.

Der Sender TOTX173 dagegen ist, abgesehen davon, dass er 5 V als Versorgungsspannung benötigt, vollständig kompatibel mit den 3,3 V vom FPGA (Abbildung 5.1) und wurde während der Entwicklungsphase dieser Thesis bereits erfolgreich mit diesem verwendet.

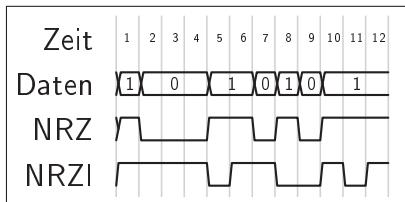
**Anmerkungen zur Datenrate (6 MBit/s vs. 12,3 MBit/s)** Laut Datenblatt erlauben sowohl der TORX173 als auch der TOTX173 eine Datenrate von 6 Mbit/s bei Verwendung von NRZ-Code<sup>4</sup> [Tos01a, S. 2][Tos01b, S. 2]

Diese Buchsen werden von WAVEFRONT (dem Hersteller der einzigen ADAT-De/Enkoder-ICs) empfohlen (s.o.), obwohl die ADAT-Datenrate bei der typischen Audio-Abtastrate von 48 kHz bereits bei

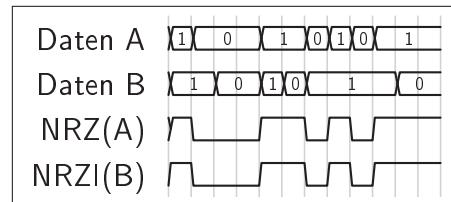
$$256 \frac{\text{Bit}}{\text{Frame}} \cdot 48.000 \frac{\text{Frame}}{\text{s}} = 12.288.000 \frac{\text{Bit}}{\text{s}}$$

also ca. 12,3 MBit/s liegt.

Die unterschiedliche Kodierung von NRZ und NRZ-I spielt dabei keine Rolle, da bei gleicher Übertragungsrate die minimale Dauer, während der das kodierte Signal gleich bleibt, identisch ist (Abbildung 5.2a). Schließlich kann ein NRZ-I Signal genau so aussehen wie ein NRZ-Signal gleicher Datenrate, sofern die Daten entsprechend gewählt werden (Abbildung 5.2b). Daher bleibt als einziger Schluss die Möglichkeit, dass diese Empfehlung sich auf empirische Testwerte von WAVEFRONT stützt, auch wenn diese Datenrate weit über den in den Datenblättern genannten Maximal-Werten liegt.



(a) Minimale Periodendauer identisch



(b) Ergebnis identisch

Abbildung 5.2: NRZ und NRZI bei gleicher Datenrate

Als Toshiba-Alternative für den 3,3 V-Einsatz (und mit einer ausreichenden Datenrate von 15MBit/s) wurden die beiden Buchsen TORX147/TOTX147 ermittelt, welche allerdings bei keinem der Distributoren auf der von Toshiba verlinkten Internetseite „Stock Check“[Tosa] [Tosb] verfügbar sind. Laut dem dort aufgeführten Distributor ARROW ELECTRONICS sind beide Buchsen von TOSHIBA abgekündigt („Discontinued/Obsolete“ [Arr14a][Arr14b]).

Die letztendlich gewählten Buchsen PLR135 und PLT133 von EVERLIGHT sind sowohl für den 3,3 Volt als auch für den 5 Volt-Betrieb ausgelegt und garantieren über den gesamten Spannungsbereich 16 MBit/s.

<sup>4</sup>NRZ (Non Return to Zero): „1“ und „0“ wird jeweils einer der Werte „High“ und „Low“ fest zugewiesen und dieser entsprechend direkt an die Leitung angelegt. Dies entspricht der „normalen“ digitalen Übertragung z.B. auf einer Platine

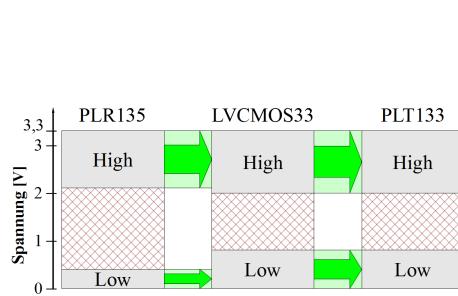


Abbildung 5.3:  
Kompatibilität der Logik-Pegel der Sender/Empfänger PLR135/PLT133 mit dem I/O-Standard LVC莫斯33 der Spartan-3A-Serie.  
Alle Pegel nach [Eve13, S. 2][Xil10b, S. 16][Eve05, S. 3] und den jeweiligen Versorgungsspannungen

Einiger Nachteil der 3,3 V-kompatiblen Empfänger TORX147 und PLR135 ist die Angabe einer minimalen Übertragungsfrequenz im Datenblatt. Wird diese unterschritten bzw. ein statisches Signal empfangen, so ist das Ausgangssignal instabil („When non-modulated signal (optical all high or all low level signal) is inputted, output signal is not stable.“[Tos06, S. 2]). Dies bedeutet, dass das ADAT-Empfangsmodul in jedem Fall eine Fehlererkennung benötigt (siehe 4.6.5.2 *Fehlererkennung*).

### 5.1.2.2 Spannungsversorgung

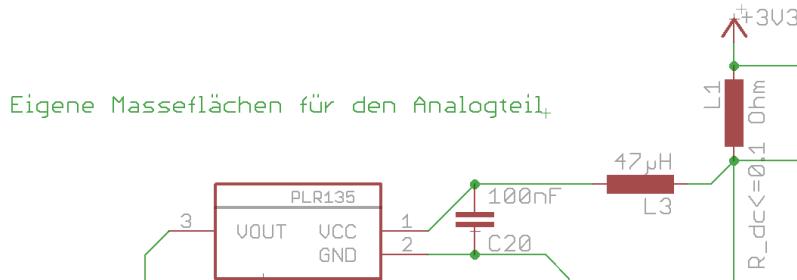


Abbildung 5.4: Spannungsversorgung der Empfangsbuchsen

Der optische Sensor in der Empfängerbuchse PLR135 benötigt eine besonders entkoppelte Spannungsversorgung. Im Datenblatt wird für  $V_{cc} = 3$  V eine Induktivität von  $47 \mu\text{H}$  zwischen dem  $V_{cc}$ -Pin der Buchse und der positiven Spannungsversorgung vorgeschlagen. Außerdem ist laut EVERLIGHT ein  $100 \text{ nF}$ -Kondensator zwischen  $V_{cc}$  und der durch die Induktivität bereits geglätteten Versorgungsspannung nötig [Eve13, S.4, „Application Circuit (1)“].

### 5.1.3 Wahl der ADAT-Taktquelle

Da ADAT-Taktgewinnung und ADAT-Dekoder getrennte VHDL-Module sind, ist es möglich den Takt aus verschiedenen Quellen zu gewinnen:

1. Eine der ADAT-Schnittstellen wird als Taktquelle benutzt  
 → Das ADAT-Gerät an der entsprechenden Schnittstelle muss auf „ADAT-Master“ gesetzt sein, alle anderen Geräte auf „ADAT-Slave“
2. Der Schaltungseigene ADAT-Oszillator wird als Taktquelle benutzt  
 → Alle ADAT-Geräte werden im „Slave“-Modus betrieben

Bei drei ADAT-Schnittstellen ergeben sich damit genau vier Taktquellen, d.h. bei Binärcodierung sind zwei digitale Eingänge zur Auswahl der Taktquelle ausreichend.

### 5.1.3.1 Das Jumper-Kreuz

Um dem Benutzer eine intuitive Einstellmöglichkeit für die Taktquelle zu geben, wird an dieser Stelle auf eine direkte Binärcodierung durch den Nutzer verzichtet. Für diese wäre auf der Platine eine 2x3-Stiftleiste nötig, bei der man die Werte der zwei Bits per Jumper zwischen VCC („High“) und GND („Low“) wechseln kann:

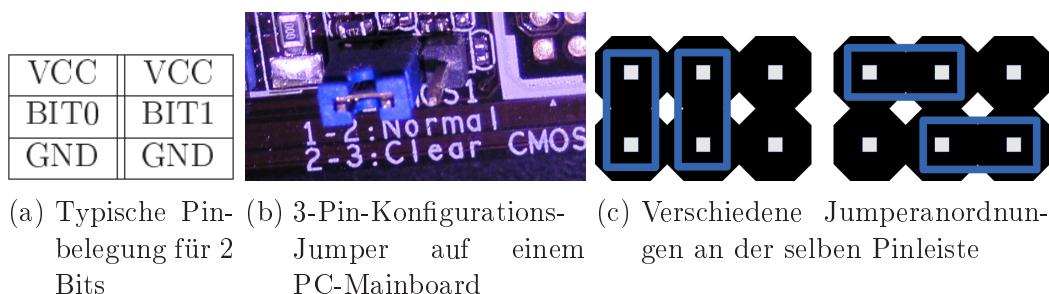


Abbildung 5.5: Die „normale“ Jumper-Konfiguration

Durch Pull-Up-Widerstände an den BIT0/BIT1-Pins könnte man mit den VCC-Pins zwei der sechs Pins einsparen (Jumper gesetzt = „Low“, Jumper nicht gesetzt = „High“), hätte aber keine Möglichkeit mehr, die nun je nach Einstellung nicht mehr benötigten Jumper trotzdem noch „richtig“ (d.h. mit zwei Pins) auf der Platine zu befestigen, damit diese nicht verlorengehen<sup>5</sup>.

Für eine Schaltung, die auch Fehlbedienungen abfängt, wäre es außerdem erforderlich, auch für den Fall, dass kein Jumper gesetzt ist, einen stabilen Betrieb der Schaltung zu gewährleisten. Das ließe sich am einfachsten durch die oben bereits erwähnten Pull-Up-Widerstände realisieren: So ist der Pin immer auf einem stabilen Potential.

Für diese Thesis wurde mit einer kreuzförmigen Pin-Anordnung ein alternativer Ansatz entwickelt. Die Möglichkeit, den Jumper praktisch wie einen Drehschalter

<sup>5</sup>Einige Elektronikhersteller verzichten auch auf diese zusätzliche Befestigungsmöglichkeit, so hat z.B. das Spartan 3A FPGA Starter-Kit von XILINX zahlreiche Jumper, die je nach Konfiguration nirgendwo „richtig“ befestigt werden können (siehe [Xil08, S.39f]).

einzusetzen, soll vor allem beim intuitiven Verständnis der Pin-Anordnung helfen. Wird eine 2-Bit Konfiguration (siehe Abbildung 5.5a) z.B. als normale zweireihige Pinleiste ausgeführt, kann der Jumper leicht auch auf ungültige Positionen gesetzt werden, da nicht direkt klar ist, ob es sich um eine 2x3 Leiste mit je zwei Steckpositionen pro Bit oder um eine 3x2 Leiste mit je einem optionalen Jumper handelt (Abbildung 5.5c). Die Kreuz-Anordnung bietet den Vorteil, dass einerseits nur ein Jumper benötigt wird, der in allen Konfigurationen an je zwei Pins befestigt ist. Sollte der Jumper verloren gehen, so wird dennoch ein gültiger Wert ausgegeben (z.B. die Standard-Konfiguration). Andererseits wird mit fünf statt sechs Pins auch noch ein Pin eingespart.

### 5.1.3.2 Ein diskreter 2-Bit-Binär-Enkoder

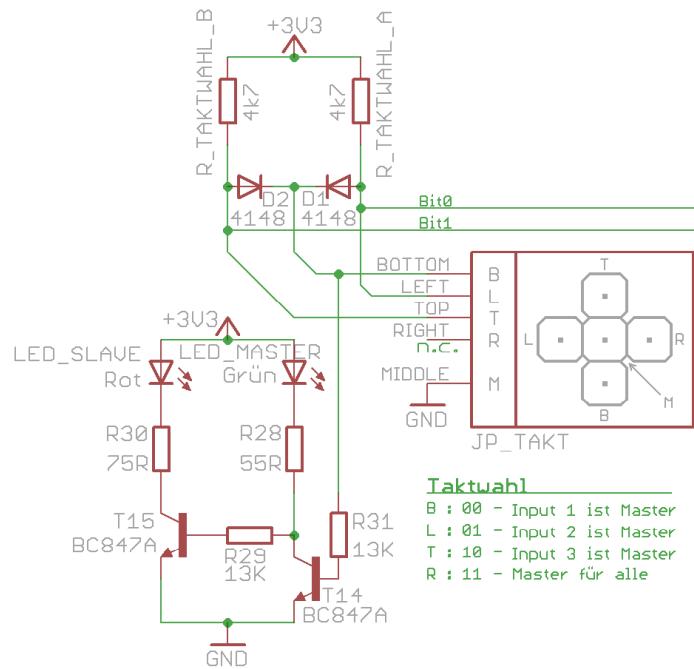


Abbildung 5.6: Die ADAT-Taktwahl

Um aus den vier Positionen des Jumpers an den Enden des Kreuzes einen binär kodierten 2-Bit-Wert zu generieren, werden lediglich die bereits erwähnten zwei Pull-Up-Widerstände und zwei zusätzliche Dioden benötigt.

Die Mitte des Kreuzes ist mit Masse verbunden. Jeder Pullup-Widerstand zieht jeweils eine Ausgangsleitung auf „High“, welche wiederum mit einem der äußeren Pin verbunden ist (in Abbildung 5.6 links und oben). Somit können bereits die

Kombinationen „01“ und „10“ erreicht werden, da der Pin am Jumper immer auf Masse gezogen wird.

Die rechte Position ist gar nicht verbunden, was der Kombination „11“ entspricht. Dies entspricht damit auch der Standardeinstellung, die vorliegt, wenn gar kein Jumper gesetzt ist.

Über zwei Dioden werden die ersten beiden Pins auf Masse gezogen, sobald der Jumper in der unteren Postion steht. Damit ist auch die Kombination „00“ abgedeckt und somit ein kompletter 2-Bit-Encoder beschrieben.

### 5.1.4 Status-LEDs

#### 5.1.4.1 Lebt die Schaltung noch?

Als wichtigstes aber sicherlich auch einfachstes Fehlererkennungselement bei der Entwicklung dient LED\_LEBTNOCH. Diese LED blinkt mit einem Takt von 1 Hz und signalisiert das der FPGA läuft, d.h. alle Spannungen sind vorhanden, der FPGA hat sich richtig konfiguriert und der ADAT-Oszillator arbeitet korrekt.

#### 5.1.4.2 ADAT-Puffer

Der größte Teil der LEDs dient zur Anzeige des ADAT-Pufferstatus. Der Puffer (siehe 4.6.8 *DynamischerADATPuffer\_BRAM*) gibt diverse Statussignale aus, um dem Benutzer/Entwickler zu signalisieren, ob, und wenn ja, welches Problem vorliegt:

LED_PUFFER_UEBERLAUF	Der Puffer ist komplett gefüllt und hat die neuen Daten nicht annehmen können
LED_PUFFER_VOLL	Der Puffer ist komplett gefüllt
LED_PUFFER_NORMAL	Der Puffer ist zum Teil gefüllt
LED_PUFFER_LEER	Der Puffer ist leer
LED_PUFFER_UNTERLAUF	Der Puffer ist leer und konnte keine Daten liefern

Zusätzlich existiert für jede ADAT-Schnittstelle noch je ein LED-Paar (LED\_FRAME\_OK und LED\_FRAME\_ERR), welches signalisiert ob der empfangene ADAT-Frame in Ordnung war oder ob es Fehler bei der Übertragung gab.

→ *Genaueres zur Fehlererkennung in Abschnitt 4.6.5.2 „Fehlererkennung“.*

#### 5.1.4.3 Wofür die Transistoren?

Theoretisch kann jeder Pin des FPGA eine LED mit 20 mA treiben. Bei insgesamt 12 über den FPGA geschalteten LEDs ergäbe das also einen theoretisch erreichbaren Gesamtstrom von 240 mA. Da die Puffer-LEDs ihren Status außerdem pro

ADAT-Frame zwei mal ändern können<sup>6</sup>, wird hier ein durchaus relevantes induktives Störpotential für die Datenleitungen in der Nähe erzeugt. Mehr dazu im Abschnitt 5.2.3 Datenleitungen. Außerdem wäre eine stärkere Störung der Versorgungsspannung des FPGAs möglich.

Um all diese Probleme zu umgehen, wird jede LED über einen eigenen Transistor vom Typ BC847A versorgt, der dann wiederum vom FPGA angesteuert wird. Bei 20 mA für eine LED und einer angenommenen Transistorverstärkung von 100 ergibt sich damit ein Basissstrom von lediglich 0,2 mA, die Datenleitungen sind also voraussichtlich sicher. Zusätzlich wird die Versorgungsspannung der LEDs noch mit einer Ferritperle sowie der Kombination aus einem Elko und einem Keramikkondensator von der 'normalen' Spannungsversorgung abgekoppelt.

Da nun jede LED ohnehin über ihren eigenen Transistor verfügt, lassen sich außerdem noch FPGA-Pins einsparen<sup>7</sup>, indem der Inverter zwischen `LED_FRAME_OK` und `LED_FRAME_ERR` außerhalb des FPGAs realisiert wird. Dafür müssen nicht einmal weitere Bauteile hinzugefügt werden:

Nehmen wir einmal an, der ADAT-Frame wurde korrekt empfangen: Der FPGA setzt seinen Ausgangspin auf „High“-Pegel und T1 schaltet durch. Jetzt leuchtet `LED_FRAME_OK`. Da die Basis von T2 nun von T1 auf Masse gezogen wird, kann T2 nicht durchschalten und `LED_FRAME_ERR` bleibt dunkel.

Tritt nun der umgekehrte Fall ein und der FPGA setzt den Pin auf „Low“, so sperrt T1. Damit wird die Basis von T2 nun über R1, `LED_FRAME_OK` und den Basiswiderstand RB2 auf 3,3 V gezogen. Der Strom durch `LED_FRAME_OK` wird dabei durch RB2 so stark begrenzt, dass diese kaum bis gar nicht leuchtet. T2 aber schaltet durch, womit nun `LED_FRAME_ERR` leuchtet und `LED_FRAME_OK` dunkel ist.

#### 5.1.4.4 $V_{CC_{INT}} = 1.2 \text{ V}$

Eine Sonderstellung nimmt die LED zur Anzeige des Ausgangsstatus des 1,2 V Spannungsreglers ein. Im Gegensatz zu den LEDs am FPGA kann hier problemlos genug Strom zur Versorgung einer LED abgezweigt werden, da keine gefährdeten

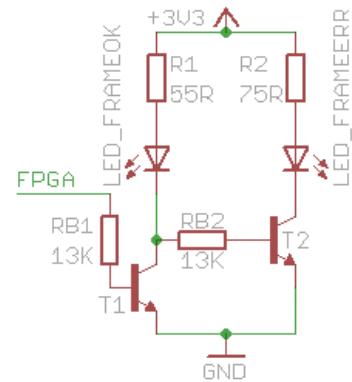


Abbildung 5.7:  
Diskreter Inverter mit Transistoren

<sup>6</sup>Am Anfang wird der Puffer gefüllt und kurz danach schon wieder ausgelesen)

<sup>7</sup>Die dann auch noch für eventuelle Erweiterungen genutzt werden könnten, ein Mangel an Pins herrscht eigentlich nicht gerade (siehe 5.2.6 Herausgeföhrte Signale).

Datenleitungen in der Nähe sind<sup>8</sup>, die zu schützen wären<sup>9</sup>. Da 1,2 V aber nicht für eine normale LED ausreichen (eine Suche beim Distributor DIGIKEY<sup>10</sup> ergab als minimal mögliche Spannung 1,6 V), wird mit dem Ausgang des Spannungsreglers zunächst ein NPN-Transistor durchgeschaltet, welcher dann die LED in seinem Kollektor-Zweig (die wiederum mit 3,3 V verbunden ist) zum Leuchten bringt.

## 5.2 Die Platine

### 5.2.0.5 Genutzte Software

Für die Erstellung von Schaltplan & Platinenlayout wurde das Programm EAGLE (Einfach Anzuwendender Grafischer Layout Editor) in der Version 6.5.0 Light verwendet. Einige nicht in der Bauteilbibliothek vorhandene Bauteile wurden selber erstellt (z.B. PLR135) oder aus anderen Projekten importiert (z.B. XC3S200A).

### 5.2.1 HF-Betrachtung nötig?

Als Faustregel gilt, dass Hochfrequenzeffekte relevant werden, sobald die Leitungslänge ca. ein Zehntel der Wellenlänge überschreitet<sup>11</sup>. Bei Digitalsignalen sollte mindestens die 3. Oberwelle, in kritischen Fällen auch noch die 5. Oberwelle betrachtet werden.

→ *Die in den folgenden Abschnitten erwähnten Leiterbahnlängen der hier vorgestellten Platine wurden mit dem „Meander“-Werkzeug von Eagle gemessen, welches eigentlich für die Erstellung von Leiterbahnen definierter Länge gedacht ist.*

#### 5.2.1.1 Die Takt-Oszillatoren

Am Beispiel des 50 MHz-Oszillators wäre die Wellenlänge

$$\lambda = \frac{c}{f} \Rightarrow \lambda = \frac{300.000.000 \frac{\text{m}}{\text{s}}}{50.000.000 \frac{1}{\text{s}}} = 6\text{m}$$

<sup>8</sup>Die oberen 10 Pinreihen des DSP-Erweiterungssteckers sind alle mit Versorgungsspannungen, Masse oder gar nicht belegt

<sup>9</sup>Abgesehen davon sind im laufenden Betrieb der Schaltung keinerlei Pegelwechsel an der LED zu erwarten, d.h. eine von dieser Leitung ausgehende Induktion findet nicht statt.

<sup>10</sup>Zum Zeitpunkt der Suche gab Digikey insgesamt 16,914 verschiedene Artikel für die Kategorie „LED Indication - Discrete“ aus. Das Ergebnis der Suche kann aufgrund der großen Datenmenge als einigermaßen repräsentativ angesehen werden.

<sup>11</sup>Dies ist kein genauer Wert, in einigen Fällen wird auch ein Sechstel der Wellenlänge erwähnt.

womit laut oben erwähnter Faustregel ab 60 cm Leitungslänge Hochfrequenzeffekte berücksichtigt werden müssten. Selbst bei Betrachtung der dritten (fünften) Oberwelle (es handelt sich hier um eine Digitalsignal) wäre die kritische Länge mit 15 cm (10 cm) weit über der tatsächlichen Leiterbahnlänge von <1 cm (siehe 5.2.2.2 *Takt*).

### 5.2.1.2 ADAT-Signale

Das ADAT Signal hat bei 48 kHz Abtastrate eine Übertragungsrate von 12.288 MBit/s. Da das Signal NRZI-Kodiert ist, der Pegel also höchstens einmal pro Bit wechselt kann, ergibt sich hierbei eine maximale Frequenz von 6,144 MHz<sup>12</sup>.

Nach Berechnung mit der dritten (fünften) Oberwelle und Berücksichtigung von  $\lambda/10$  sollte für die ADAT-Signale also die Leitungslänge unter

$$\frac{300.000.000 \frac{\text{m}}{\text{s}}}{4 \cdot 6.144.000 \frac{1}{\text{s}}} \cdot \frac{1}{10} = 1,2 \text{m}$$

(5. Oberwelle: 0,8 m) bleiben. Das lässt sich absolut problemlos einhalten: Die längste ADAT-Leitung auf der Platine ist nur 10,6 cm lang.

### 5.2.1.3 McBSP

Das höchste Leitungslängen-Frequenz-Produkt tritt beim McBSP auf. Dort müssen die Daten von allen ADAT-Schnittstellen zusammen übertragen werden. Inklusive der Erweiterung der Userbits auf ein 24 Bit-Element (siehe 3.4.2 *Auffüllen auf 24 Bit pro Element*) ergibt sich eine Datenrate von

$$\frac{3 \cdot 9 \cdot 24 \text{Bit}}{\text{Frame}} \cdot \frac{48.000 \text{Frame}}{\text{s}} = 31 \text{MBit/s}$$

Der dazugehörige Takt auf den CLK-Leitungen beträgt demnach 31 MHz (eine Periode pro Bit), womit für die Leitung ab

$$\frac{300.000.000 \frac{\text{m}}{\text{s}}}{4 \cdot 31.000.000 \frac{1}{\text{s}}} \cdot \frac{1}{10} = 0,24 \text{m}$$

eine Hochfrequenzbetrachtung nötig wäre (5. Oberwelle: 16 cm). Die mit 11,5 cm längste Leitung (CLKS) hält – trotz des unschönen Verlaufs einmal um den Stecker J3 herum – hier noch ausreichend Abstand zum kritischen Bereich der dritten Oberwelle, so dass auch die Restlänge der Leitung, welche auf dem DSP-Board weiterläuft, noch abgedeckt sein sollte.

---

<sup>12</sup>Bei einem Datensignal mit lauter Einsen wechselt der Signalpegel mit jedem Bit. Die Periode des resultierenden Rechtecksignals beträgt dabei also 2 Bit-Längen.

Bei der Betrachtung der 5. Oberwelle sieht es schon etwas knapper aus: Ab dem Ende der Leitung wären gerade einmal 3,5 cm für Stecker und Leitung zum DSP verfügbar.

Die Leitung im Stecker wird bei dem in [Spe06b, S. 3-3] erwähnten Platinenabstand bei zusammengesteckten Boards von 0.465 Zoll (etwa 1,2 cm) in etwa genau so lange sein. Die CLKS0-Leitung läuft auf dem DSP-Board über eine ca. 9,3 mm lange Leitung<sup>13</sup> direkt zu einem 8-Fach-Bus-Switch-IC. Im weiteren Verlauf ist die Leitung durch Abschlusswiderstände bereits angepasst [Spe06c, S. 11], die kritische Leiterbahnlänge von DSP-Board und Stecker beträgt also großzügig gerundet etwa 2,2 cm.

Das IC hat einen Übergangswiderstand von 5 Ohm [Tex97, S. 1], im Vergleich zum Widerstand der restlichen Leiterbahn – laut [Mik13] ist bei  $35\mu\text{m}$  Kupferdicke und einer Leiterbahnbreite von 10 mil (0,254 mm) mit  $20,3 \text{ m}\Omega/\text{cm}$ , also in etwa  $0,28 \Omega$  für die ganze Leitung zu rechnen – ist die Leitung also hier effektiv beendet, da eventuelle reflektierte Signalanteile im  $5\Omega$ -Widerstand massiv gedämpft werden würden.

Die ganze zu betrachtende Leitung hat demnach eine Länge von ca. 13,7 cm. Folglich sind auch für die 5. Oberwelle keine Hochfrequenzeffekte zu erwarten.

## 5.2.2 Der FPGA

Zentrale Steuereinheit auf der Platine ist der FPGA, der aus diesem Grund auch in etwa mittig auf der Platine platziert ist.

### 5.2.2.1 Sinnvolle Platzierung von Abblockkondensatoren

Zentral für das korrekte Funktionieren von Digitalschaltungen ist die sinnvolle Platzierung von Abblockkondensatoren. Als Grundregel gilt dabei „100 nF an jedes VCC-GND-Paar“.

XILINX stellt auf seiner Internetpräsenz unter anderem eine Checkliste zum Platinendesign zur Verfügung. Dort wird als Anhaltspunkt das Verteilen von Keramikkondensatoren mit Werten bis zu 100nF auf die einzelnen VCC-GND-Paare genannt [Xil14]. Der Abstand zu den FPGA Pins soll dabei maximal 1–2 cm betragen.

In Abbildung 5.8 ist das resultierende Layout zu sehen. Um die Kondensatoren möglichst nahe am FPGA zu platzieren, sind sie direkt unter diesem angebracht und in der Mitte verbunden, wo auch die verschiedenen Spannungsversorgungen ankommen (in Abbildung 5.8 jeweils die von links oben kommenden breiten Leiterbahnen).

Dies bietet außerdem den Vorteil, dass die „normalen“ Pins auf dem oberen Layer ohne Hindernisse nach außen geroutet werden können.

<sup>13</sup>Diese Länge wurde anhand eines Abgleichs der in [Spe06b, S. B-2] gezeigten Übersichtszeichnung mit dem Leiterbahnverlauf im Platinenlayout [Spe06a, S. 9] ermittelt.

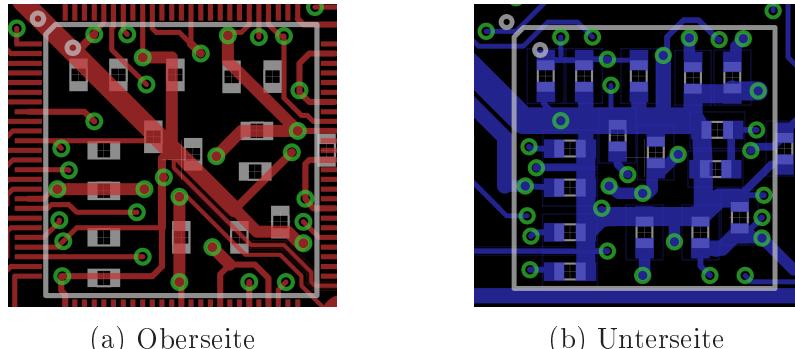


Abbildung 5.8: Abblockkondensatoren am FPGA (Ansicht von oben)

Um die Abblockkondensatoren mit einer möglichst geringen Impedanz anzubinden, sind die Stromversorgungsleitungen entsprechend breit ausgelegt. Die Masseleitung bildet dabei anstatt eines Sternpunktes ein Rechteck (Abbildung 5.8b), um auch in der Mitte noch Kondensatoren anbringen zu können.

### 5.2.2.2 Takt

Insbesondere der Takt sollte möglichst einwandfrei am FPGA ankommen. Aus diesem Grund wurden die dafür vorgesehenen Oszillatoren möglichst nahe am FPGA platziert (Leitungslänge jeweils 6-7 mm) und das Layout dem „Recommended land pattern“ inklusive des dort empfohlenen 10 nF-Kondensators aus dem Datenblatt [Abr11, S. 2] nachempfunden.

### 5.2.3 Datenleitungen

Grundsätzlich sollten Leitungen, die sich gegenseitig nicht beeinflussen sollen, nie parallel verlegt werden, um ein kapazitive bzw. induktive Einkopplung des Signals zu vermeiden. Sollte dies nicht vermeidbar sein, ist der Abstand möglichst groß zu wählen.

In Abbildung 5.9a war an einer Stelle auf der Platine weder das parallele Verlegen der Leitungen vermeidbar, noch ein größerer Abstand möglich.

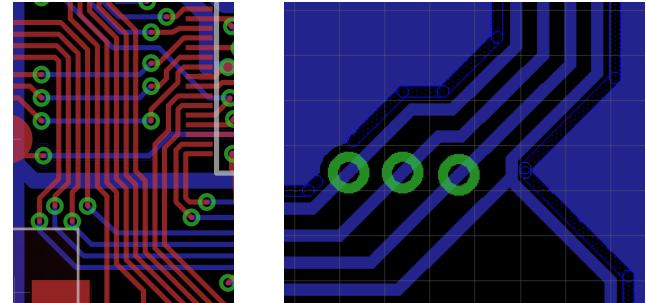
Es handelt sich dabei einerseits um die Leitungen zum McBSP und andererseits um die zu den Anzeige-LEDs. Die McBSP-Signale sollten dabei auf keinen Fall gestört werden.

Um die induktive Kopplung der LED-Signale auf die McBSP-Leitungen zu minimieren, muss der Strom durch die potentiellen „Stör-Leitungen“ möglichst gering sein. Daher werden die LEDs (20 mA pro LED) in diesem Fall nicht direkt vom FPGA-Pin getrieben sondern über Transistoren geschaltet (siehe 5.1.4.3 *Wofür die*

*Transistoren?*). So ist an der kritischen Stelle der Strom (und damit die Induktive Störung) um einige Größenordnungen geringer (ca. 0,2 mA).

Abbildung 5.9b zeigt die Leitungen zu den ADAT-Buchsen. Um hier eine unnötig große kapazitive Kopplung mit der Massefläche zu verhindern (welche zu einer unerwünschten Abrundung des Signalverlaufs führen würde), wurde auf dem „bRestrict“-Layer in Eagle ein Rahmen (im Bild gepunktet) um diese Leitungen gezeichnet, so dass Eagle die Massefläche nicht ganz so nah an die Datenleitungen legen kann.

Andererseits sind die Datenleitungen hier weiter von einander entfernt als auf der restlichen Platine, was die Kopplung der Signale untereinander verringert.



(a) Verschiedene Datenleitungen neben  
parallel der Massefläche

Abbildung 5.9: Problematische Datenleitungen

### 5.2.4 Separate Analog-Flächen für die ADAT-Buchsen

Bei den ADAT-Buchsen ist eine sorgfältige Behandlung der Spannungsversorgung besonders wichtig:

Auf der Platine sind separate Analog-Flächen (AVCC/AGND) gemäß der Application Note am Ende des PLR135-Datenblattes [Eve13, S. 8] über Ferritperlen an die normale Spannungsversorgung angeschlossen. Die einzelnen Buchsen wiederum sind erst nach je einem 100 nF Abblock-Kondensator (und beim Empfänger zusätzlich einer 47  $\mu$ H-Induktivität) mit diesen Analog-Flächen verbunden, damit sie sich auch nicht gegenseitig stören können.

### 5.2.5 Kühlung des Spannungsreglers

Der 1,2 V-Spannungsregler LD1117 ist ein klassischer Linearregler, regelt also den Ausgang auf 1,2 V und wandelt den Rest der Eingangsspannung in Wärme um.

Da bei dem hier zu erwartenden ICCINT-Maximalstrom von etwa 350 mA damit bereits 0,735 W verheizt werden<sup>14</sup>, ist eine Kühlung des Spannungsreglers in Betracht zu ziehen:

Laut Datenblatt hat der LD1117 im SOT-223-Gehäuse vom einen Bauteil-Luft-Wärmewiderstand von  $R_{thJA} = 110^{\circ}\text{C}/\text{W}$ . [ST-13, S.7, Tabelle 2]

<sup>14</sup> $P_{verheizt} = 0,35\text{A} \cdot (3,3\text{V} - 1,2\text{V}) = 0,735\text{W}$

Damit ergibt sich bei einer Raumtemperatur von 20°C bereits eine Erwärmung des Spannungsreglers auf ca. 100°C:

$$t_{LD1117} = 20^\circ\text{C} + 110^\circ\text{C}/\text{W} \cdot 0,735\text{W} = 100,85^\circ\text{C}$$

Diese Temperatur verkraftet der LD1117 zwar (er ist noch bis 125°C innerhalb der „Absolute Maximum Ratings“ [ST-13, S.7, Tabelle 1]), ein versehentliches Anfassen des Spannungsreglers würde bei dieser Temperatur aber bereits zu Verbrennungen führen. Außerdem gilt in der Elektronik die Faustregel, dass sich pro 10°C Erwärmung die Lebensdauer der Bauteile halbiert. Eine niedrige Temperatur ist also auch im Sinne der Haltbarkeit der Schaltung erstrebenswert.

Da beim SOT-223-Gehäuse die Kühlfläche nur direkt auf die Platine gelötet werden kann (im Gegensatz zum TO-220-Gehäuse, bei dem hier die Befestigung eines Kühlkörpers möglich ist), muss die Platine als Kühlkörper dienen oder die Wärme zumindest bis zu diesem weiterleiten.

### 5.2.5.1 Experimentelle Bestimmung des Wärmewiderstands einer Kupferfläche als Kühlkörper

Da die Angaben diverser Halbleiterhersteller über die nötige bzw. angemessene Platinenfläche pro verheiztem Watt weit auseinandergehen, wurde eine eigene Messreihe aufgenommen:

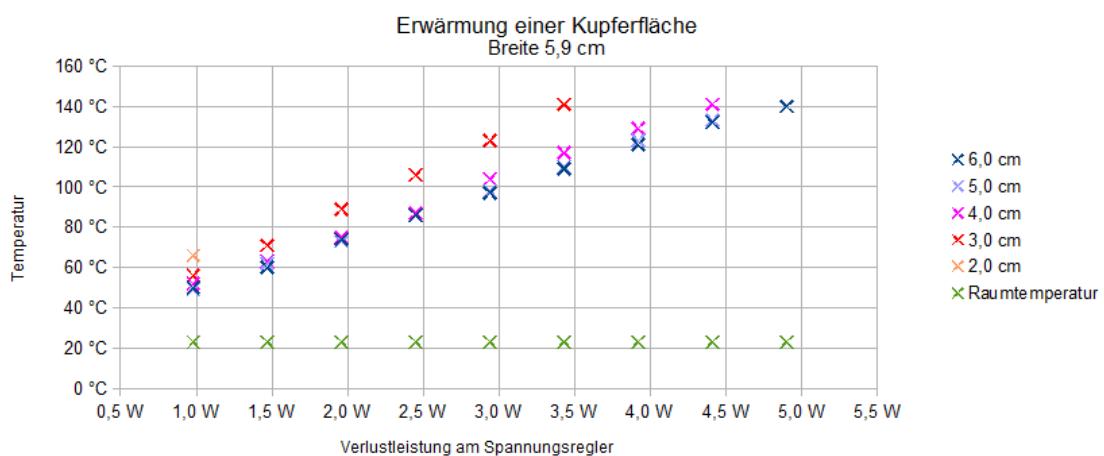


Abbildung 5.10: Verlustleistung vs. Temperatur

Dazu wurde ein 5 V-Spannungsregler im TO-220-Gehäuse auf ein Stück Platine aufgelötet. Zusammen mit einem 10-Ohm Leistungswiderstand als Last ist der Spannungsregler nun einem konstanten Strom von 0,5 A (in der realen Messung 0,49 A)

ausgesetzt. Misst man nun die Spannung zwischen Ein- und Ausgang des Reglers, so lässt sich daraus die im Regler in Wärme umgewandelte Leistung berechnen.

Im Versuch wurde nun der Spannungsabfall über dem Regler in 1 V-Schritten vorgegeben<sup>15</sup> und nach ca. 10 Minuten der Temperaturwert abgelesen. Der Temperatursensor war dabei direkt an der Lötstelle angebracht.

Mit jeder Messreihe wurde die Länge der Platine um einen Zentimeter gekürzt. Wie in Abbildung 5.10 zu sehen ist, nimmt die Effektivität der Kühlung von einer 6 cm langen Platine ausgehend zunächst kaum ab. Erst ab 3 cm Länge ist ein deutlicher Unterschied zu bemerken.

Bei der 2 cm-Messung ist nur der erste Wert vorhanden: Beim Messen des zweiten Wertes begann der Spannungsregler massiv zu Oszillieren und auch der erste Wert war nicht mehr wiederholbar<sup>16</sup>.

Dennoch ist bereits hier ein deutlicher Trend zu erkennen. Mit jedem Zentimeter weniger steigt die Temperatur deutlich schneller. Berechnet aus den Messreihen jeweils den Mittelwert der Wärmewiderstände und setzt diesen in Beziehung zur verbleibenden Länge der Platine, so erhält man die in Abbildung 5.11 zu sehende Kurve. Der Wärmewiderstand nimmt demnach in etwa exponentiell mit der Platinenlänge ab.

### 5.2.5.2 Vergleich mit der Theorie

Von jedem Stück Platine kann einerseits ein gewisser Wärmeanteil an die Luft abgegeben werden (in Abb. 5.12 ist der dazugehörige Wärmewiderstand jeweils vertikal dargestellt). Andererseits wird aber auch ein Teil über den Wärmewiderstand der Platine an das nächste Platinenstückchen weitergeleitet (im Bild horizontal).

Demnach ergibt sich im elektrischen Ersatzschaltbild eine exponentiell abnehmende Wertigkeit der einzelnen Platinenabschnitte bei der Kühlung. In Abbildung 5.12

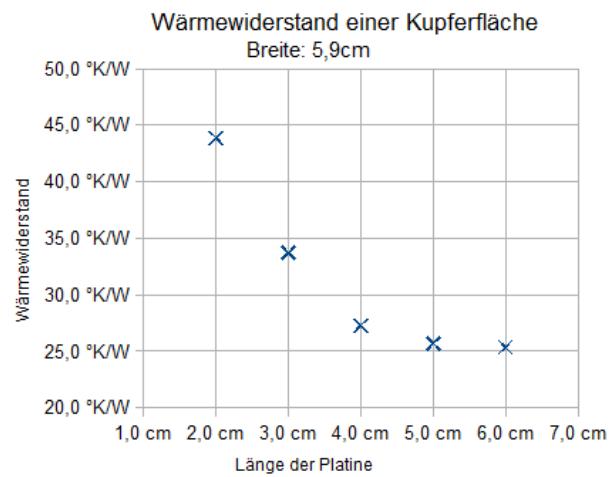


Abbildung 5.11: Wärmewiderstand vs. Platinenlänge

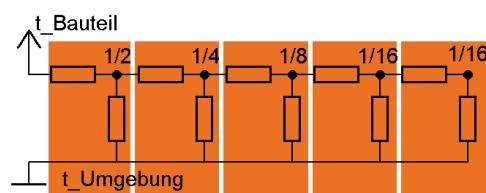


Abbildung 5.12:  
Aufteilung der Wärme auf die Wärmewiderstände der Platinenabschnitte

<sup>15</sup>Dabei muss die Versorgungsspannung natürlich über 5V liegen. Sinnvolle Werte ergaben sich ab ca. 7 V.

<sup>16</sup>Statt 66°C ergaben sich nur noch 55°C, was sogar weniger wäre als bei der 3 cm-Platine. Vermutlich wurde der Spannungsregler durch das mehrfache Überschreiten der 125°C-Grenze (typischerweise die maximale Betriebstemperatur von Halbleitern laut Datenblatt) dauerhaft beschädigt.

wurde beispielhaft mit gleichen Widerständen für beide Übergänge gerechnet.

### 5.2.5.3 Anwendung auf die konkret realisierte Platine

Die auf der Platine realisierte Kühlfläche hat eine Fläche von ca. 6 cm<sup>2</sup>. Über thermische Vias wird allerdings auch die Unterseite der Platine zur Kühlung genutzt. Beide Seiten zusammen ergeben 12 cm<sup>2</sup>, entsprechen also in etwa den 11,8 cm<sup>2</sup> der 2 cm-Platine. Mit ca. 44 °K/W ergäben sich also nun etwa 52°C, eine Temperatur die durchaus akzeptabel ist:

$$t_{LD1117,12\text{cm}^2\text{Kupfer}} = 20^\circ\text{C} + 44^\circ\text{K}/\text{W} \cdot 0,735\text{W} = 52,34^\circ\text{C}$$

Dabei ist zu beachten das die untere Seite nicht ganz so viel Wärme abgeben wird, da sich die Wärme zwischen DSP-Platine und ADAT-Platine voraussichtlich zu einem gewissen Grad stauen wird.

Andererseits ist die Anbindung der Platinenfläche hier besser als im Experiment: Dort musste die Wärme von der dünnen Kupferlage noch weit an den Rand geleitet werden, wenn auch dieser einen Kühleffekt haben sollte. Hier sind die Strecken deutlich kürzer, da sich die zwei halb so großen Kupferflächen beide direkt am Spannungsregler befinden.

Insgesamt sollte die Kupferfläche also voraussichtlich auch bei der Maximalbelastung ausreichen. Zur Sicherheit wurden dennoch zwei Befestigungslöcher und ein vom Lötstopplack freier Bereich in der Nähe des Spannungsreglers vorgesehen, um im Zweifelsfall noch einen zusätzlichen Kühlkörper befestigen zu können.

### 5.2.6 Herausgeführte Signale

Um eventuelle spätere Erweiterungen der Platine zu ermöglichen, werden alle ungenutzten Pins des FPGA auf zwei 20-Pol-Wannensteckern herausgeführt (jeweils mit ein mal 3,3 V und GND als Referenzpegel). Die Pinbelegung ist im Anhang zu finden (siehe E.3 *Herausgeführte FPGA-Pins*).

Außerdem werden auch die ADAT-Signale (wieder inklusive 3,3 V) von und zu den Buchsen herausgeführt.

Als Programmierschnittstelle ist JTAG in der 14-Poligen Wannenstecker-Konfiguration nach [Xil07b, S. 2] herausgeführt.

### 5.2.7 Testpunkte

Um im Fehlerfall gute Diagnosemöglichkeiten zu haben, wurden zahlreiche Testpunkte auf der Platine angebracht. Dies betrifft vor allem die Versorgungsspannungen, welche an verschiedenen Stellen abgegriffen werden. So gibt es einerseits

Testpunkte für GND und VCC (3,3 V) am Spannungsregler, wenn sie also gerade die Schaltung erreicht haben, andererseits aber auch Messmöglichkeiten für die Signale direkt unter dem FPGA.

Um VCC möglichst genau in der Mitte des FPGAs messen zu können (wo alle VCC-Leitungen zusammentreffen), musste die dazugehörige Messleitung mit Hilfe der tRestrict-Ebene von Eagle explizit von der VCC-Fläche (oberes Platinenlayer) freigestellt werden (Abbildung 5.13 - die Leitung verläuft vom Testpunkt unten rechts nach links oben, in Richtung der Mitte des FPGA).

Der links daneben abgebildete (dunkelrote) Testpunkt greift 1,2 V aus der Mitte des FPGA ab. Da keine 1,2 V-Fläche vorliegt, ist hier kein explizites Freistellen nötig gewesen.

Als weiterer wichtiger Punkt können außerdem GND und VCC einmal kurz vor dem Übergang auf die AGND bzw. AVCC-Fläche (siehe 5.2.4 *Separate Analog-Flächen für die ADAT-Buchsen*) und einmal auf dieser Fläche gemessen werden.

Als letzte Messmöglichkeit gibt es schließlich noch Testpads an den Oszillatoren. Diese sind jedoch deutlich kleiner, um das Taktsignal nicht durch unnötig große Kapazitäten zu beeinflussen.

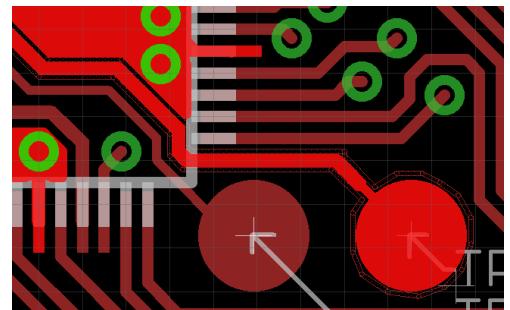


Abbildung 5.13:  
VCC-Testpunkt mit tRestrict (Die hellroten Flächen haben VCC-Potential)

# 6 Analyse

## 6.1 ADAT-Loopback

Aufgrund der erwähnten Probleme beim DSP wurde zum Testen aller anderen entwickelten Module ein ADAT-Loopback aufgebaut: Von einem Audiointerface wird ein ADAT-Signal an die FPGA-Platine gesendet, dort dekodiert, in den internen Puffer geschrieben und kodiert wieder an die Wandlerkarte zurückgesendet.

Da das Taktgewinnungsmodul zwar erfolgreich simuliert wurde, praktisch aber aufgrund der Timings im FPGA noch nicht korrekt funktioniert (eine dazugehörige Timing-Constraint wird nicht eingehalten), arbeitet der FPGA hier als ADAT-Master.

Da das verwendete Audiointerface „ADI-8 AE“ sein Synchronisationssignal wider erwarten nicht mit dem vom FPGA gesendeten Synchronisationssignal synchronisierte, musste kurzerhand noch eine einfache Synchronisationserkennung implementiert werden. (*Da diese nach dem selben Prinzip arbeitet wie die Synchronisationserkennung im Taktgewinnungsmodul und als einzige Abweichung mit dem bereits vorhandenen ADAT-Takt getaktet wird, wurde sie hier nicht weiter behandelt*).

Abschließend erfolgte eine Latenzmessung mit einem 100Hz-Sinussignal:

Die Latenz des Gesamtsystems inklusive Audiointerface beträgt 2 ms. Der ADAT-Loopback war in der Beispieldmessung mit 40  $\mu$ s daran beteiligt. Aufgrund der nicht vorhandenen Synchronisation der Synchronisationssignale kann dieser Wert allerdings von Messung zu Messung von etwas mehr als einem bis zu zwei ADAT-Frames schwanken (In Abbildung 6.2 sind es fast zwei Frames).

In der Simulation mit synchronen Synchronisierungssignalen ergab sich dagegen immer eine Latenz von 2 ADAT-Frames.

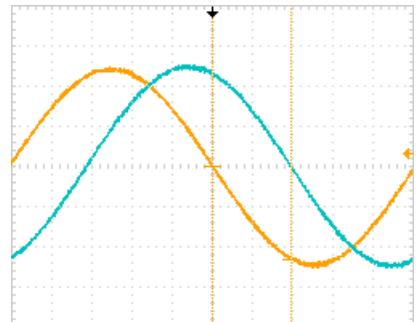


Abbildung 6.1:  
Messung der Latenz inkl. dem Audiointerface - y:200 mV/Div; x:1 ms/Div

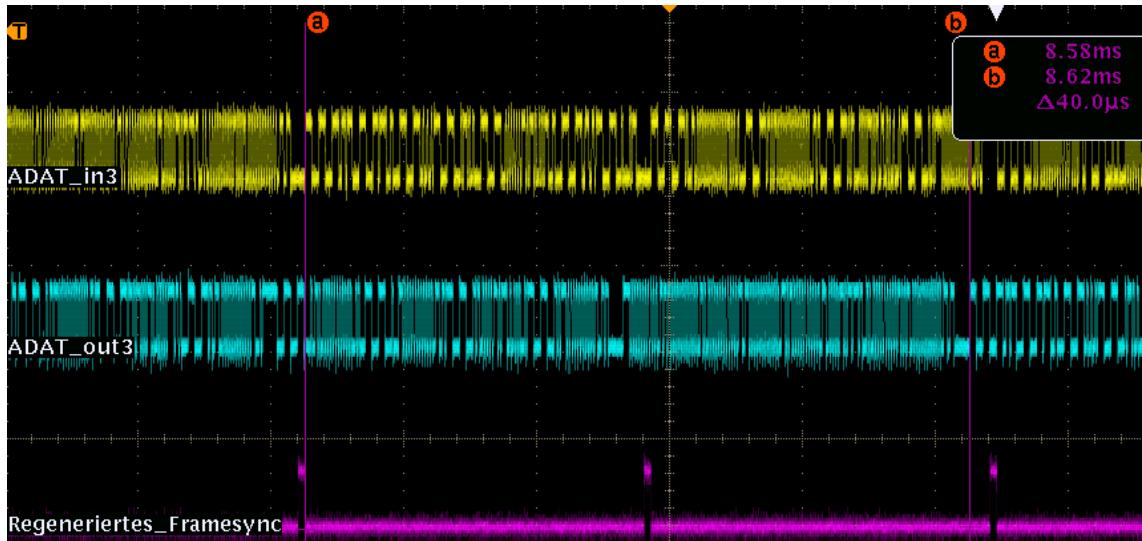


Abbildung 6.2: Messung der ADAT-Latenz - y:5 V/DIV; x:20  $\mu$ s/DIV

## 6.2 Kommunikation über den McBSP

In praktischen Tests konnten bereits erfolgreich Daten an den DSP gesendet werden. Dieser sendete allerdings nicht wie vorgesehen die selben Daten zurück, so dass eine Ausgabe von sinnvollen ADAT-Daten nicht möglich war. In der Simulation des McBSP\_Interface-Moduls verlief die Kommunikation dagegen absolut problemlos.  
→ Aufgrund der vielen Daten-Ein- und Ausgänge im FPGA wird an dieser Stelle auf eine grafische Darstellung der McBSP-Simulation verzichtet.

## 6.3 Taktgewinnung

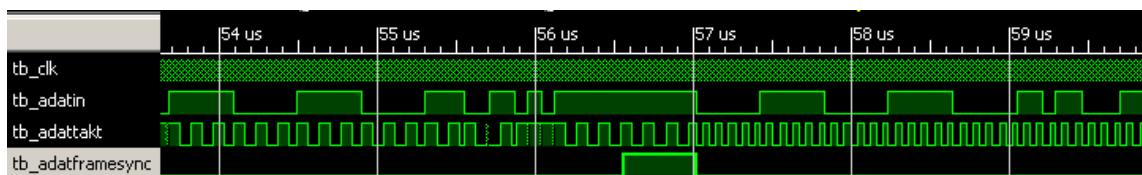


Abbildung 6.3: Simulation der Taktgewinnung - Übergang vom instabilen zum stabilen ADAT-Takt

Da die Taktgewinnung, wie bereits erwähnt, in der Praxis aufgrund von Timingproblemen im FPGA noch nicht funktioniert, kann hier nur auf die Simulation eingegangen werden:

Nach der eingestellten Anzahl von Signaltransistionen (hier 200) ist das FrameSync-Signal stabil und wechselt am nächsten erkannten Synchronisierungsmuster im Eingangssignal kurz auf „High“ um am Ende des Synchronisierungsmusters wieder auf „Low“ zu fallen. Einen ADAT-Frame nachdem Frame-Sync wieder auf „Low“ gewechselt hat, ist auch das Taktsignal stabil.

# 7 Schlussbetrachtung

## 7.1 Fazit

Die DSP-Verbindung über den McBSP konnte leider nie erfolgreich getestet werden, da der DSP die empfangenen Daten nie korrekt zurückschickte.

Auch wenn die Entwicklung und Simulation der VHDL Module absolut problemlos verlief, so war doch die Umsetzung in einen realen FPGA wider erwarten der größte Arbeitsaufwand. Einige Probleme konnten durch die Verwendung der aufgezeichneten ADAT-Sequenzen in den VHDL-Testbenches allerdings auch schon im Vorfeld ausgeräumt werden.

Mit der Entscheidung, beim Platinenentwurf auf einen größeren FPGA im BGA-Gehäuse zu verzichten, musste das Design plötzlich massiv optimiert werden, um in den gewählten kleineren FPGA zu passen. Durch die Verwendung von Block-RAM und mit dem mit der Zeit immer besseren Verständnis für die nötige Logik-Denkweise bei der VHDL-Entwicklung konnten letztlich sogar drei ADAT-Schnittstellen realisiert werden.

Damit wäre die in der Aufgabenstellung geforderte Berücksichtigung der „Ausweitung auf bis zu drei ADAT-Kanäle“ bereits erledigt. Da bei der Entwicklung aber konsequent auf die Verwendung von `Generics` geachtet wurde, ist mit einem entsprechend größeren FPGA aber letztlich ein Ausbau auf beinahe beliebige Kanalzahlen möglich. Alternativ könnten durch eine leichte Modifizierung des Designs (Multiplexen der McBSP-Verbindung) aber auch mehrere kleine (und damit leichter lötbare) FPGAs verwendet werden.

## 7.2 Ausblick

Da – wie erst zum Schluss festgestellt wurde – die Synchronität der Synchronisationssignale bei ADAT keineswegs immer gegeben ist (lediglich der Takt wird immer synchronisiert), und die angeschlossenen Audiowandler das Audiosignal damit nicht gleichzeitig Samplen würden, sollte eine weitere Synchronisierungsmöglichkeit für problematische Geräte ergänzt werden. Die Implementierung eines Wordclock-Ausgangs ist in VHDL zwar in wenigen Zeilen geschrieben<sup>1</sup>, da der Ausgangspegel dabei allerdings 5 V sein sollte [Aud11, S. 14], ist dies zukünftigen Revisionen der Platine vorbehalten.

---

<sup>1</sup>Das Wordclocksignal entspricht einem einfachen Rechtecksignal mit der Audio-Abtastfrequenz, also dem durch 256 geteilten ADAT-Bit-Takt.

# A Literatur

- [Ada] Website. Abgerufen am 25.11.2013 um 12:21. URL: [http://ringbreak.dnd.utwente.nl/~mrjb/hd24tools/techdocs/Adat\\_sync\\_pinout.txt](http://ringbreak.dnd.utwente.nl/~mrjb/hd24tools/techdocs/Adat_sync_pinout.txt).
- [Abr11] Abracon Corporation (Hrsg.) *3.3V HCMOS / TTL Compatible SMD Crystal Clock Oscillator. ASFL1*. Datenblatt. Rancho Santa Margarita, CA, Apr. 2011.
- [Ale] Alesis (Hrsg.) *adat. 8 Track Professional Digital Audio Recorder*. o.J. [vermutl. 1991/92]. Cumberland, RI.
- [Ale01a] Alesis (Hrsg.) *Alesis ADAT HD24 Reference Manual*. Los Angeles, CA, 2001.
- [Ale01b] Alesis (Hrsg.) *Alesis ADAT Proprietary Multichannel Optical Digital Interface. Addendum February 2001 2X Sample Rate (96kHz) Specification*. Version 1.0. Cumberland, RI, Feb. 2001.
- [Ale01c] Alesis Studiosound GmbH (Hrsg.) *Alesis ADAT HD24 Bedienungsanleitung*. Willich, 2001.
- [Ard05] John Ardizzoni. *A Practical Guide to High-Speed Printed-Circuit-Board Layout*. In: Analog Dialogue 39-9. Sep. 2005.
- [Arr14a] Arrow Electronics. *TORX147 / Arrow Electronics Components Search*. Website. abgerufen am 08.01.2014 um 19:33. Jan. 2014. URL: <http://components.arrow.com/part/search/TORX147>.
- [Arr14b] Arrow Electronics. *TOTX147 / Arrow Electronics Components Search*. Website. abgerufen am 08.01.2014 um 19:28. Jan. 2014. URL: <http://components.arrow.com/part/search/TOTX147>.
- [Ash08] Peter J. Ashenden. *The Designer's Guide to VHDL*. 3. Aufl. San Francisco, CA: Morgan Kaufmann, Mai 2008. ISBN: 978-0-12-088785-9.
- [Aud11] RME Audio. *ADI-8 DS. Bedienungsanleitung*. Haimhausen, März 2011.
- [Bar+94] Keith Barr u. a. *Method and apparatus for providing a digital audio interface protocol*. Patent. US 5297181. März 1994.
- [Bar+99] Keith Barr u. a. *Method for synchronizing digital audio tape recorders*. Patent. Version B1. EP 0621976 B1. Juni 1999.

- [Ell13] Rod Elliott. *ESP - Heatsink design and transistor mounting*. Website. Version Rev.11. abgerufen am 17.01.2014 um 17:10. März 2013. URL: <http://sound.westhost.com/heatsinks.htm>.
- [Eve05] Everlight (Hrsg.) *Technical Data Sheet Photolink- Fiber Optic Transmitter. PLT133/T*. Datenblatt. Version 2. New Taipei City, Juli 2005.
- [Eve13] Everlight (Hrsg.) *Photolink- Fiber Optic Receiver PLR135/T*. Datenblatt. Version 4. New Taipei City, Mai 2013.
- [Kle13] Christian Kleinhennrich. *Besprechung zu den Zielen der Thesis*. persönliches Gespräch. Wuppertal, Okt. 2013.
- [Mik13] Mikrocontroller.net. *Leiterbahnbreite*. abgerufen am 18.01.2014 um 16:47. Dez. 2013. URL: <http://www.mikrocontroller.net/wikisoftware/index.php?title=Leiterbahnbreite&oldid=79926>.
- [NTi12] NTi Audio (Hrsg.) *ADAT*. Application Note. Schaan, Liechtenstein, Jan. 2012.
- [RS13] Jürgen Reichardt und Bernd Schwarz. *VHDL-Synthese. Entwurf digitaler Schaltungen und Systeme*. 6. Aufl. München: Oldenbourg Verlag, 2013. ISBN: 978-3-486-71677-1.
- [ST-13] ST-Microelectronics (Hrsg.) *LD1117. Adjustable and Fixed Low Drop Positive Voltage Regulator*. Datenblatt. Version 33. Genf, Nov. 2013.
- [Spe06a] Spectrum Digital (Hrsg.) *6455\_dsk\_gerber*. Teil der „DSK6455/EVM6455 product CD“. Stafford, TX, Juni 2006.
- [Spe06b] Spectrum Digital (Hrsg.) *TMS320C6455 DSK. Technical Reference*. Teil der „DSK6455/EVM6455 product CD“. Stafford, TX, Juni 2006.
- [Spe06c] Spectrum Digital (Hrsg.) *TMS320C6455 DSK*. Teil der „DSK6455/EVM6455 product CD“. Stafford, TX, Juni 2006.
- [Tex97] Texas Instruments (Hrsg.) *SN74CBTLV3245A. Low-Voltage Octal FET Bus Switch*. Datenblatt. Revision Aug. 2005. Dallas, TX, Juli 1997.
- [Tex06a] Texas Instruments (Hrsg.) *TMS320C6000 DSP Multichannel Buffered Serial Port ( McBSP). Reference Guide*. Version G. Dallas, TX, Dez. 2006.
- [Tex06b] Texas Instruments (Hrsg.) *TMS320C6455 Chip Support Library. API Reference Guide*. Dallas, TX, Mai 2006.
- [Tex13] Texas Instruments (Hrsg.) *Xilinx - Spartan-3A - XC3S\_A Series - Processor Power Reference Design*. Website. abgerufen am 08.01.2014 um 14:05. Dallas, TX, 2013. URL: <http://www.ti.com/analog/docs/refdesignovw.tsp?familyId=64&contentType=2&genContentId=34821>.

- [Tos01a] Toshiba (Hrsg.) *TORX173. Fiber Optic Receiving Module*. Datenblatt. Minato, Japan, Aug. 2001.
- [Tos01b] Toshiba (Hrsg.) *TOTX173. Fiber Optic Transmitting Module*. Datenblatt. Minato, Japan, Aug. 2001.
- [Tos06] Toshiba (Hrsg.) *TORX147PL(F,T). Fiber Optic Receiving Module*. Datenblatt. Minato, Japan, Dez. 2006.
- [Tosa] Toshiba. *Stock Check „TORX147“*. Website. abgerufen am 08.01.2014 um 19:24, eingegebener Suchbegriff: „torx147“. URL: <http://www.stkcheck.com/evs/toshiba/toshheader.asp>.
- [Tosb] Toshiba. *Stock Check „TOTX147“*. Website. abgerufen am 08.01.2014 um 19:25, eingegebener Suchbegriff: „totx147“. URL: <http://www.stkcheck.com/evs/toshiba/toshheader.asp>.
- [Wav05a] Wavefront Semiconductor (Hrsg.) *AL1401AG ADAT® Optical Encoder*. Datenblatt. Cumberland, RI, Sep. 2005.
- [Wav05b] Wavefront Semiconductor (Hrsg.) *AL1402G ADAT® Optical Decoder*. Datenblatt. Cumberland, RI, Sep. 2005.
- [Wav] Wavefront. *Wavefront Semiconductor / Products / Product Lineup*. Website. o.J. [vermutl. 2013]. abgerufen am 08.01.2014 um 23:04. URL: <http://www.wavefrontsemi.com/index.php?products>.
- [Xil05] Xilinx (Hrsg.) *XAPP463 Using Block RAM in Spartan-3 Generation FPGAs*. Version 2.0. San Jose, CA, März 2005.
- [Xil07a] Chris Zeh (Hrsg.: Xilinx). *WP257 What Are Period Constraints?* Version 1.0. San Jose, CA, März 2007.
- [Xil07b] Jameel Hussein (Hrsg.: Xilinx). *XAPP986 Bulletproof Configuration Guide for Spartan-3A FPGAs*. Version 1.0.2. San Jose, CA, Nov. 2007.
- [Xil08] Xilinx (Hrsg.) *UG334 Spartan-3A/3AN FPGA Starter Kit Board User Guide*. Version 1.1. San Jose, CA, Juni 2008.
- [Xil09] Xilinx (Hrsg.) *UG332 Spartan-3 Generation Configuration User Guide. Extended Spartan-3A, Spartan-3E, and Spartan-3 FPGA Families*. Version 1.6. San Jose, CA, Okt. 2009.
- [Xil10a] Xilinx (Hrsg.) *DS123 Platform Flash In-System Programmable Configuration PROMs*. Version 2.18. San Jose, CA, Mai 2010.
- [Xil10b] Xilinx (Hrsg.) *DS529 Spartan-3A FPGA Family: Data Sheet*. Version 2.0. San Jose, CA, Aug. 2010.

- [Xil10c] Xilinx (Hrsg.) *UCF Editing*. Version 12.1 rev1. Präsentation zum Training-Kurs „Advanced FPGA Implementation“. San Jose, CA, 2010. URL: [http://xilinx.eetrend.com/files-eetrend-xilinx/forum/201103/1746-3205-02a\\_ucf\\_editing\\_12.pdf](http://xilinx.eetrend.com/files-eetrend-xilinx/forum/201103/1746-3205-02a_ucf_editing_12.pdf).
- [Xil10] Stephanie Tapp (Hrsg.: Xilinx). *XAPP 951 Configuring Xilinx FPGAs with SPI Serial Flash*. Version 1.3. San Jose, CA, Sep. 2010.
- [Xil11] Xilinx (Hrsg.) *UG331 Spartan-3 Generation FPGA User Guide. Extended Spartan-3A, Spartan-3E, and Spartan-3 FPGA Families*. Version 1.8. San Jose, CA, Juni 2011.
- [Xil12] Xilinx (Hrsg.) *UG612 Timing Closure User Guide*. Version 14.3. gültig für ISE Design Suite 14.3 bis 14.6. San Jose, CA, Okt. 2012.
- [Xil13a] Xilinx (Hrsg.) *UG613 Spartan-3A and Spartan-3A DSP Libraries Guide for HDL Designs*. Version 14.5. gültig für ISE Design Suite 14.5 bis 14.6. San Jose, CA, März 2013.
- [Xil13b] Xilinx (Hrsg.) *UG625 Constraints Guide*. Version 14.5. gültig für ISE Design Suite 14.5 bis 14.6. San Jose, CA, Apr. 2013.
- [Xil13c] Xilinx (Hrsg.) *UG627 XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices*. Version 14.5. gültig für ISE Design Suite 14.5 bis 14.6. San Jose, CA, März 2013.
- [Xil14] Xilinx (Hrsg.) *PCB Design Checklist. Checklist to help PCB and system designers complete a PCB*. Website. abgerufen am 18.01.2014 um 11:30. 2014. URL: [http://www.xilinx.com/products/design\\_resources/signal\\_integrity/si\\_pcbcheck.htm](http://www.xilinx.com/products/design_resources/signal_integrity/si_pcbcheck.htm).

## B Abbildungsverzeichnis

2.1	Frameaufbau ADAT . . . . .	6
2.2	Taktgewinnung ADAT-Patent . . . . .	9
2.3	Figur 7 aus dem ADAT-Patent [Bar+94] . . . . .	10
2.4	Die Synchronisierungssequenz in einem realen ADAT-Signal . . . . .	12
2.5	Synchronisierungssequenz NRZI-Kodiert . . . . .	12
2.6	Abtast-Takt . . . . .	13
3.1	Verbindung des FPGAs mit dem DSP . . . . .	16
3.2	Frameaufbau McBSP . . . . .	17
3.3	„Data Delay“ im DSP . . . . .	17
3.4	Simulation des Sendevorgangs vom FPGA zum DSP . . . . .	20
4.1	Das Programm „Tektronix CSV -> VHDL Testbench“ . . . . .	35
4.2	Übersicht über das Zusammenspiel der VHDL-Module . . . . .	45
5.1	Logik-Pegel der Sender/Empfänger TOTX/TORX-173 . . . . .	60
5.2	NRZ und NRZI bei gleicher Datenrate . . . . .	61
5.3	Logik-Pegel der Sender/Empfänger PLR135/PLT133 . . . . .	62
5.4	Spannungsversorgung der Empfangsbuchsen . . . . .	62
5.5	Die „normale“ Jumper-Konfiguration . . . . .	63
5.6	Die ADAT-Taktwahl . . . . .	64
5.7	Diskreter Inverter mit Transistoren . . . . .	66
5.8	Abblockkondensatoren am FPGA . . . . .	70
5.9	Problematische Datenleitungen . . . . .	71
5.10	Verlustleistung vs. Temperatur . . . . .	72
5.11	Wärmewiderstand vs. Platinenlänge . . . . .	73
5.12	Aufteilung der Wärme auf die Wärmewiderstände der Platinenabschnitte . . . . .	73
5.13	VCC-Testpunkt mit tRestrict . . . . .	75
6.1	Messung der Latenz inkl. dem Audointerface . . . . .	76
6.2	Messung der ADAT-Latenz . . . . .	77
6.3	Simulation der Taktgewinnung . . . . .	77

E.1	Spannungsversorgung . . . . .	86
E.2	Takt-Oszillatoren . . . . .	86
E.3	Auswahl der Konfigurationsquelle . . . . .	86
E.4	FPGA und PROM . . . . .	87
E.6	Nicht verwendete FPGA-Pins . . . . .	88
E.5	ADAT-Taktwahl & Konfigurationsquellenauswahl . . . . .	88
E.7	ADAT-Buchsen . . . . .	89
E.8	Verbindungsstecker zum DSP . . . . .	90
E.9	Anzeige-LEDs . . . . .	91
E.10	Das Platinenlayout . . . . .	92
E.11	Restliche FPGA-Pins . . . . .	93
E.12	Testpunkte auf der Platine . . . . .	94
E.13	Taktrouting im FPGA . . . . .	95

## C Tabellenverzeichnis

2.1	Bittypen im ADAT-Frame . . . . .	6
2.2	Die ADAT-Userbits . . . . .	6
4.1	Gehäuseformen der Spartan-3A-Serie . . . . .	24
4.2	Einige Datentypen . . . . .	27
4.3	Vordefinierte Zeit-Suffixe in VHDL . . . . .	33
4.4	Funktion <code>AdatDatenArray_TO_Bitvector</code> : Bitanordnung . . . . .	47

# D Listings

4.1	Entity-Beispiel . . . . .	25
4.2	Architecture-Beispiel . . . . .	26
4.3	Initialisierung von Variablen und Signalen . . . . .	27
4.4	Die Range-Angabe . . . . .	28
4.5	Typdefinitionen und Enumerationstypen . . . . .	28
4.6	Prozesse in VHDL . . . . .	29
4.7	Komponentendeklaration . . . . .	30
4.8	Instanziierung von Komponenten . . . . .	31
4.9	Der „Generate“-Befehl . . . . .	31
4.10	Packages in VHDL . . . . .	32
4.11	Zeitliche Verzögerung mit <b>after</b> . . . . .	33
4.12	CSV-Datei des Tektronix DPO4034-Oszilloskops . . . . .	34
4.13	Ausgabe von „Tektronix CSV -> VHDL Testbench“ . . . . .	36
4.14	Constraints: <b>CONFIG PART</b> . . . . .	37
4.15	Constraints: <b>CONFIG VCCAUX</b> . . . . .	38
4.16	Constraints: <b>TIMESPEC</b> und <b>PERIOD</b> . . . . .	38
4.17	Constraints: <b>NET</b> . . . . .	38
4.18	Constraints: <b>NET</b> -Pinzuweisung . . . . .	38
4.19	Constraints: Kommentare in .ucf-Dateien . . . . .	39
4.20	Constraints im VHDL-Code . . . . .	39
4.21	Ressourcenoptimierung am Beispiel der NRZI-Dekodierung . . . . .	42
4.22	Definition der Konstante <b>ADAT_DATENMENGE</b> . . . . .	45
4.23	Definition des Datentyps <b>ADAT_Daten</b> . . . . .	46
4.24	Definition des Datentyps <b>ADAT_Daten_Array</b> . . . . .	46
4.25	Definition des Datentyps <b>ADAT_Frame_Array</b> . . . . .	46
4.26	Deklaration der Funktion <b>AdatDatenArray_TO_Bitvector</b> . . . . .	47
4.27	<b>ADAT_Taktgewinnung</b> - Die Entity . . . . .	48
4.28	<b>ADAT_Dekoder</b> - Die Entity . . . . .	49
4.29	<b>ADAT_Enkoder</b> - Die Entity . . . . .	50
4.30	<b>ADAT_BLOCKRAM</b> - Die Entity . . . . .	52
4.31	<b>DynamischerADATPuffer_BRAM</b> - Die Entity . . . . .	53
4.32	<b> McBSP_Interface</b> - Die Entity . . . . .	56

# E Anhang

## E.1 Schaltplan

Alle Vorwiderstände sind auf normale 20mA-LEDs ausgelegt

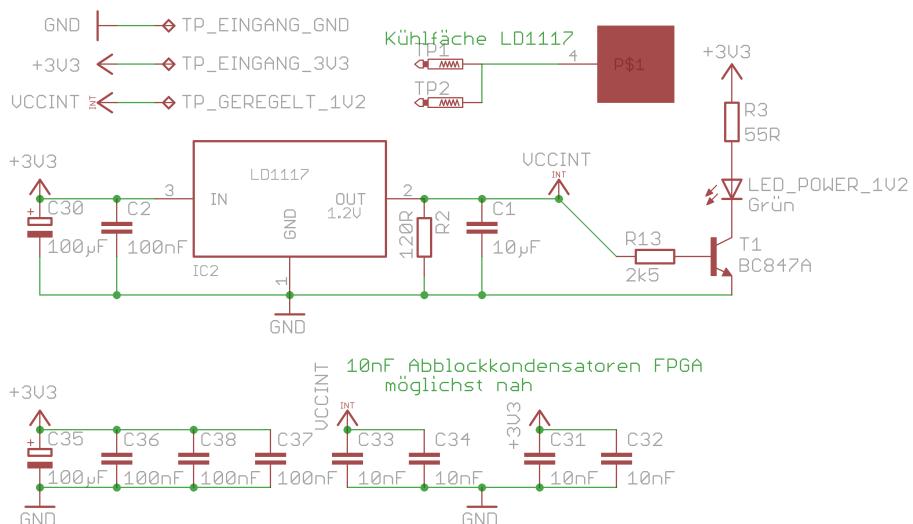


Abbildung E.1: Spannungsversorgung

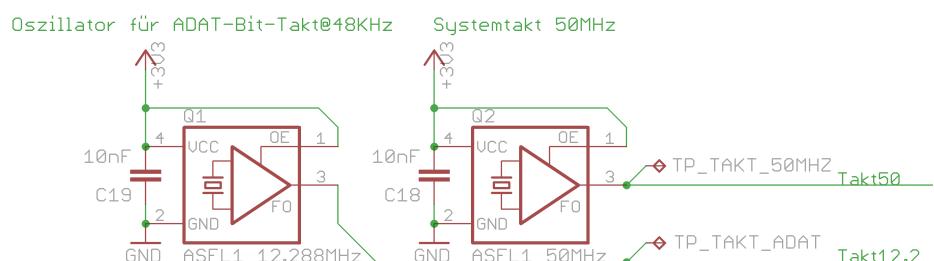


Abbildung E.2: Takt-Oszillatoren



Abbildung E.3: Auswahl der Konfigurationsquelle

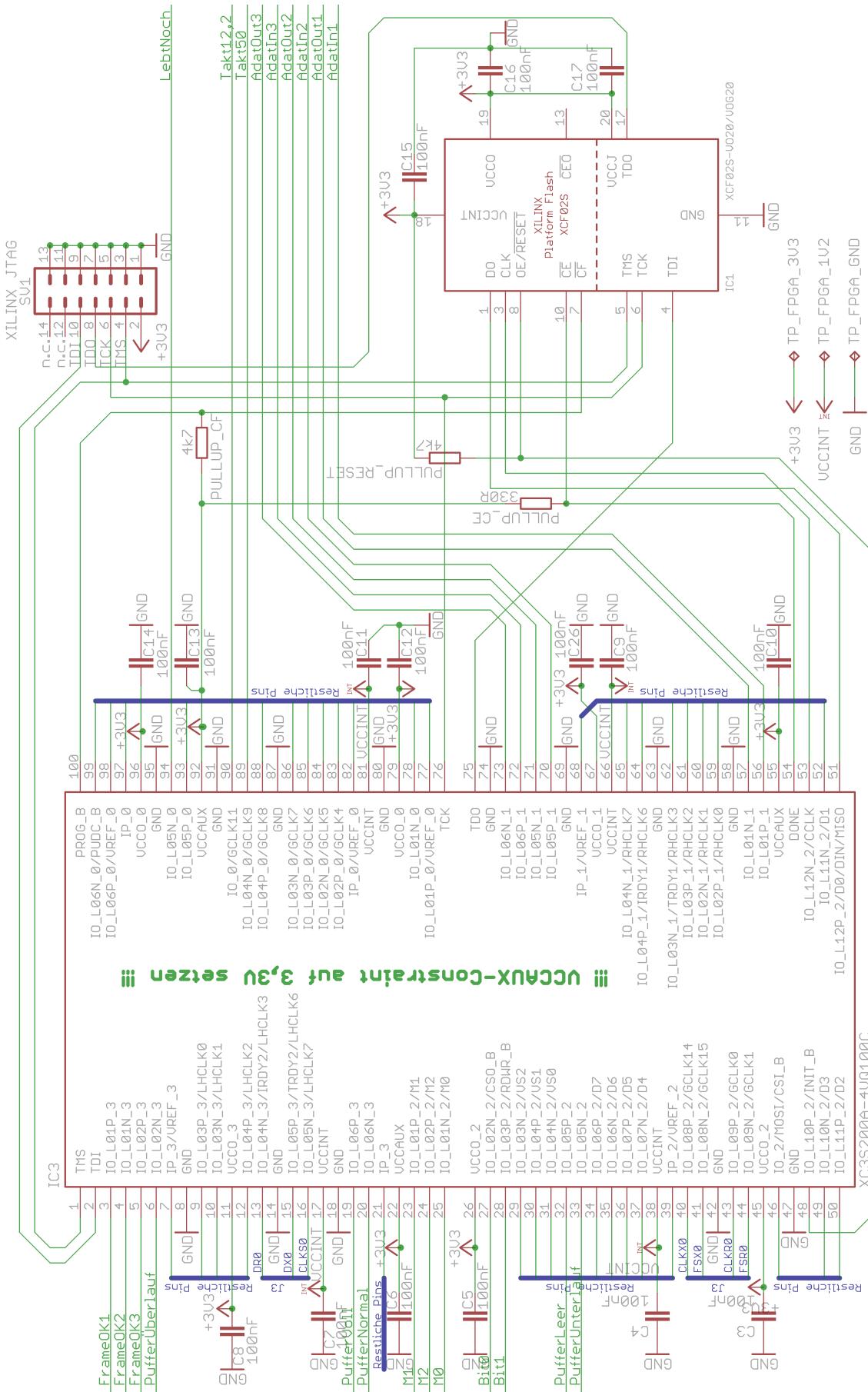


Abbildung E.4: FPGA und PROM

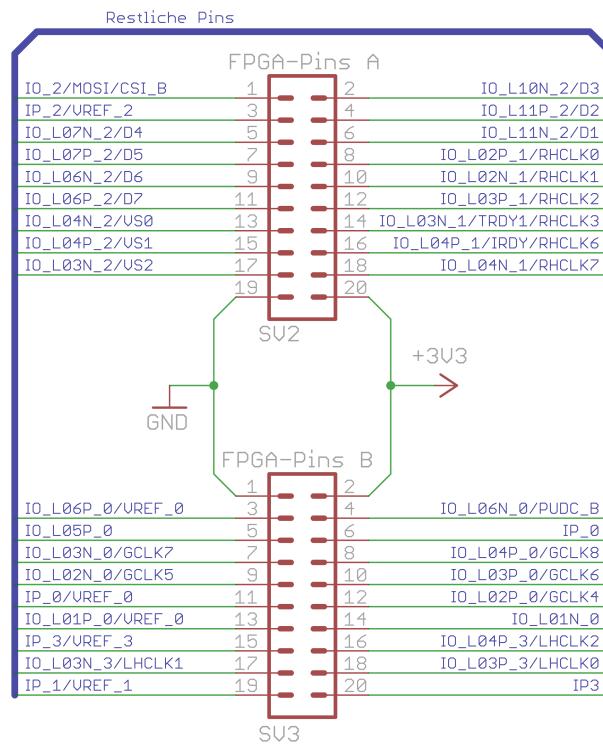


Abbildung E.6: Nicht verwendete FPGA-Pins

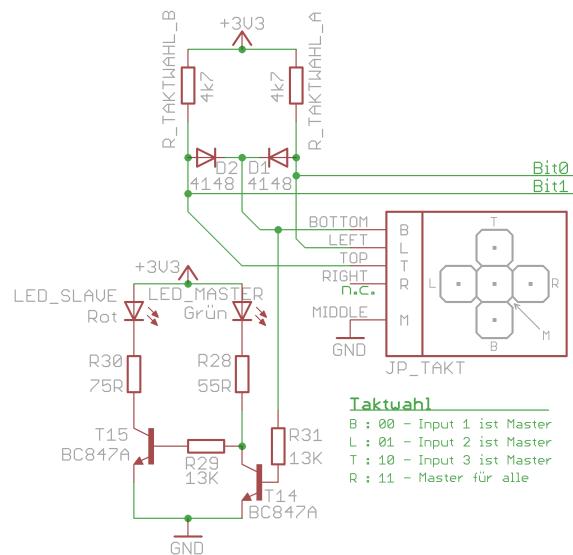


Abbildung E.5: ADAT-Taktwahl & Konfigurationsquellenauswahl

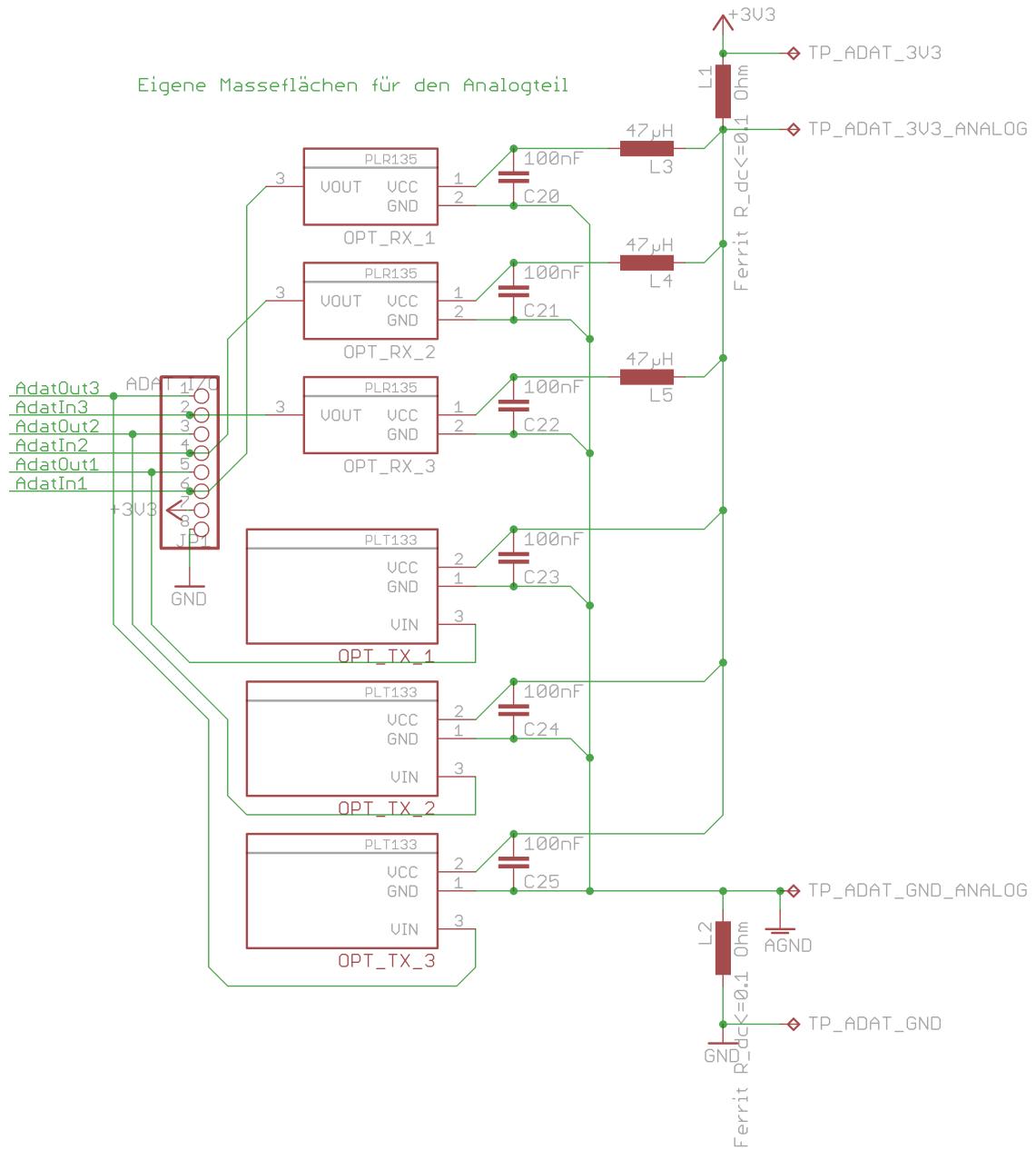


Abbildung E.7: ADAT-Buchsen

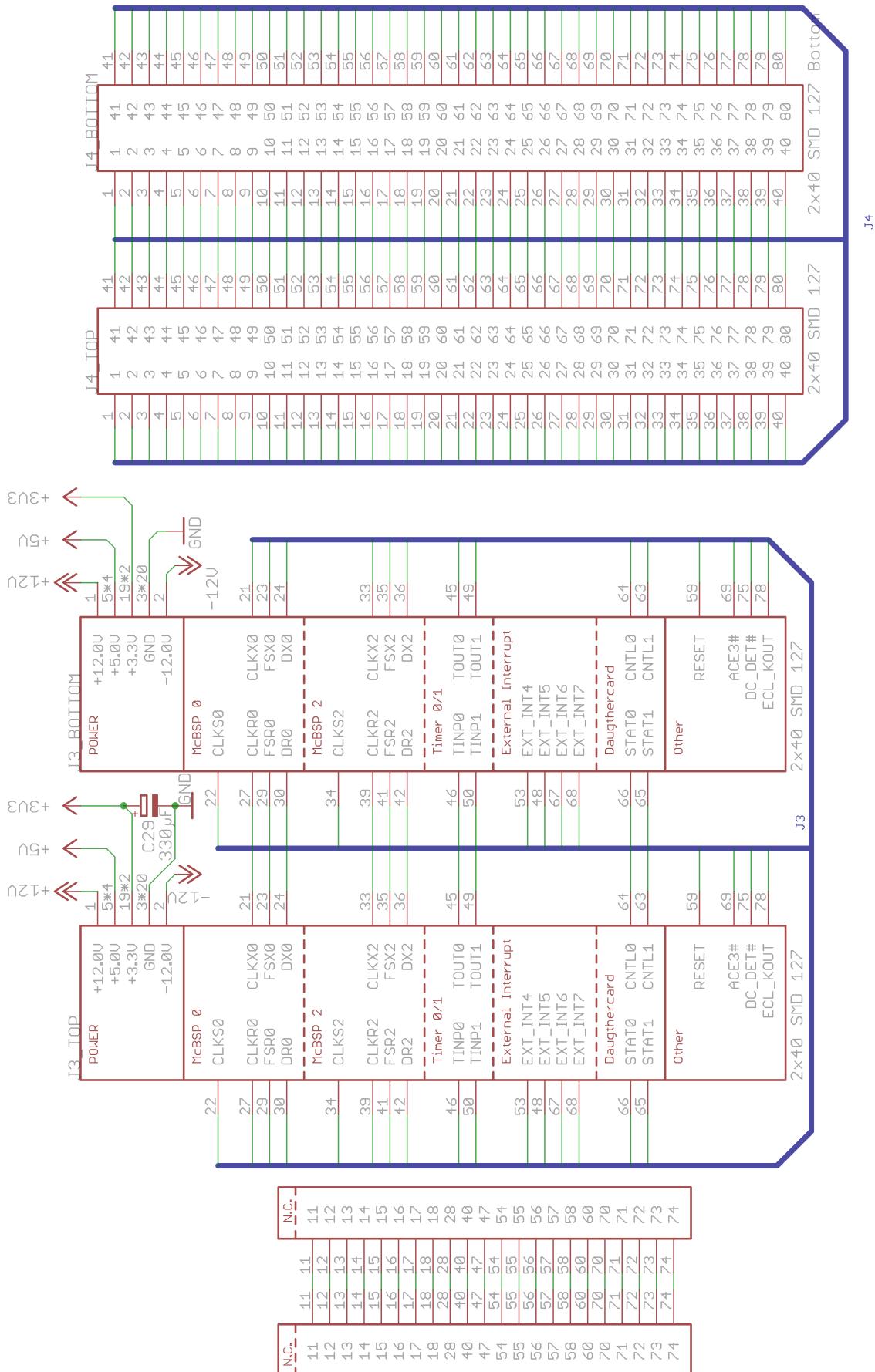


Abbildung E.8: Verbindungsstecker zum DSP

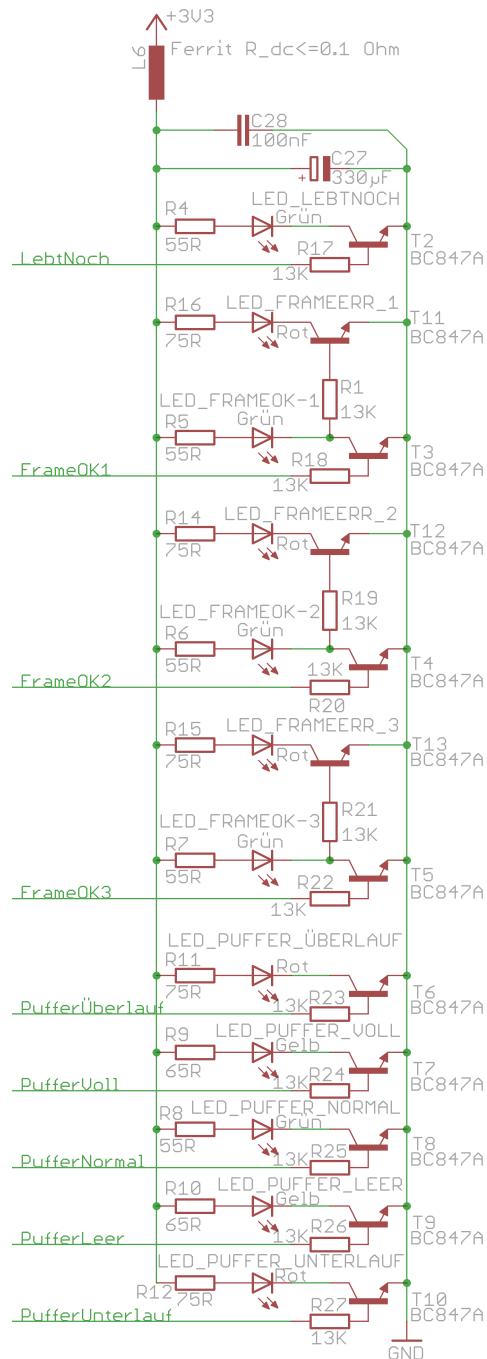


Abbildung E.9: Anzeige-LEDs

## E.2 Platinenlayout

→ Die ADAT-Sender PLT133 sind leider Spiegelverkehrt eingeplant worden, müssen für die korrekte Funktion aber lediglich auf der Unterseite der Platine montiert werden.

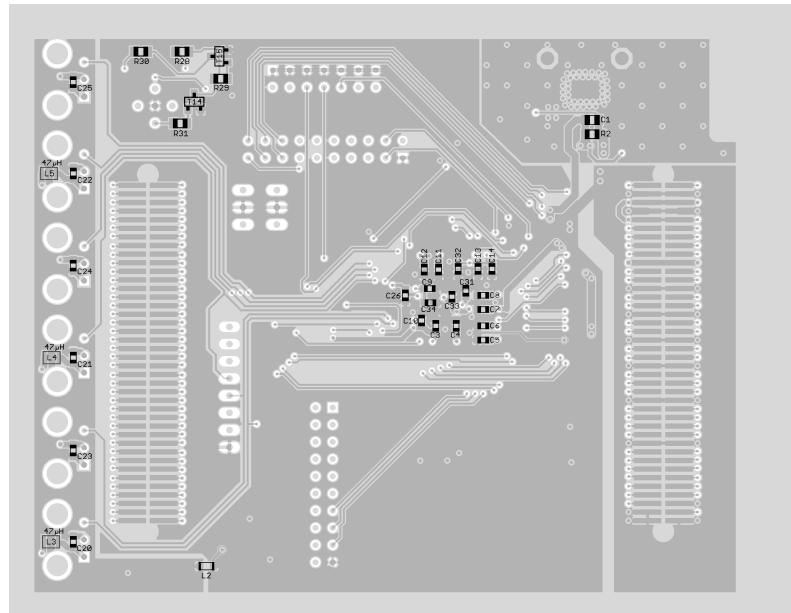
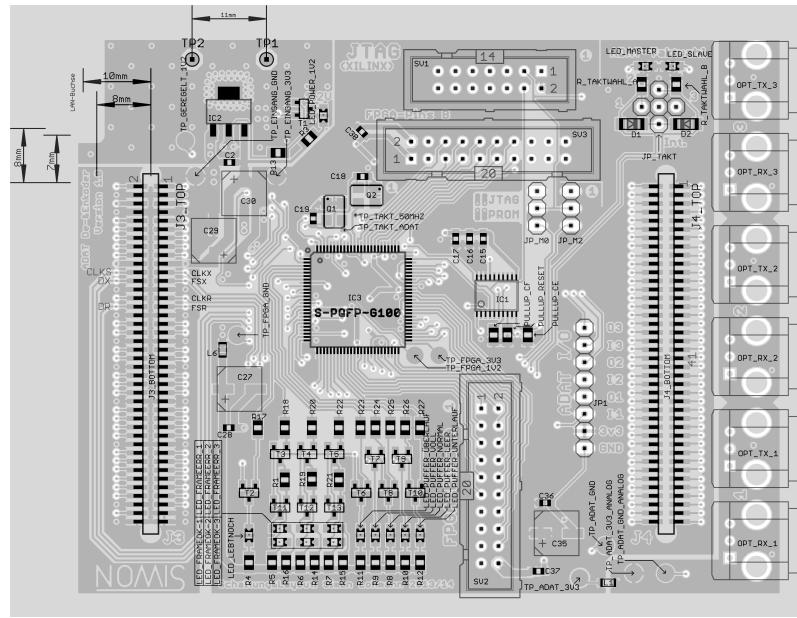


Abbildung E.10: Das Platinenlayout

### E.3 Herausgeführte FPGA-Pins

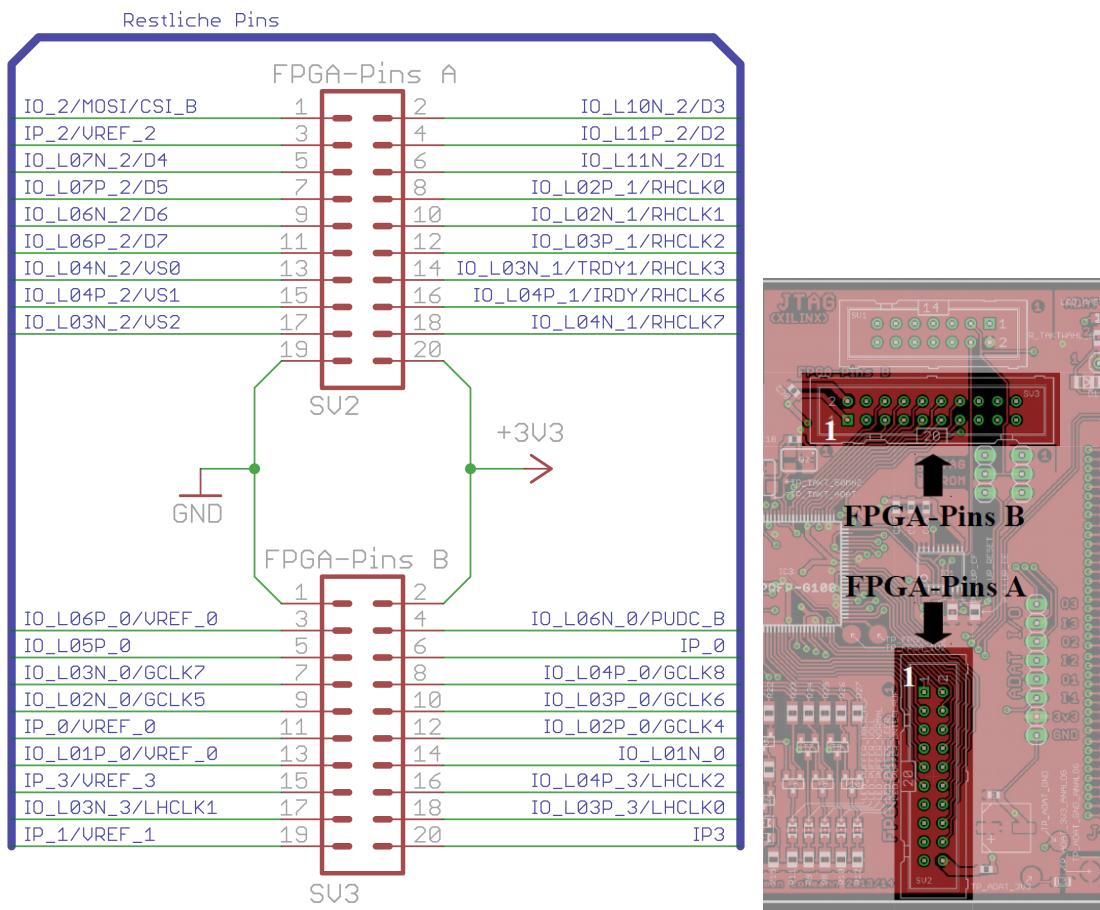


Abbildung E.11: Restliche FPGA-Pins

Für eine eventuelle Erweiterbarkeit sind alle nicht benutzten FPGA-Pins direkt auf normale Wannenstecker herausgeführt.

## E.4 Testpunkte

Auf der Platine sind zahlreiche Testpunkte verteilt. Die FPGA-Testpunkte gehen direkt unter dem FPGA an den entsprechenden Sternpunkt.

Alle Testpunkt-Spannungen werden möglichst nahe am zu messenden Punkt abgegriffen und von dort zum Testpunkt weitergeleitet.

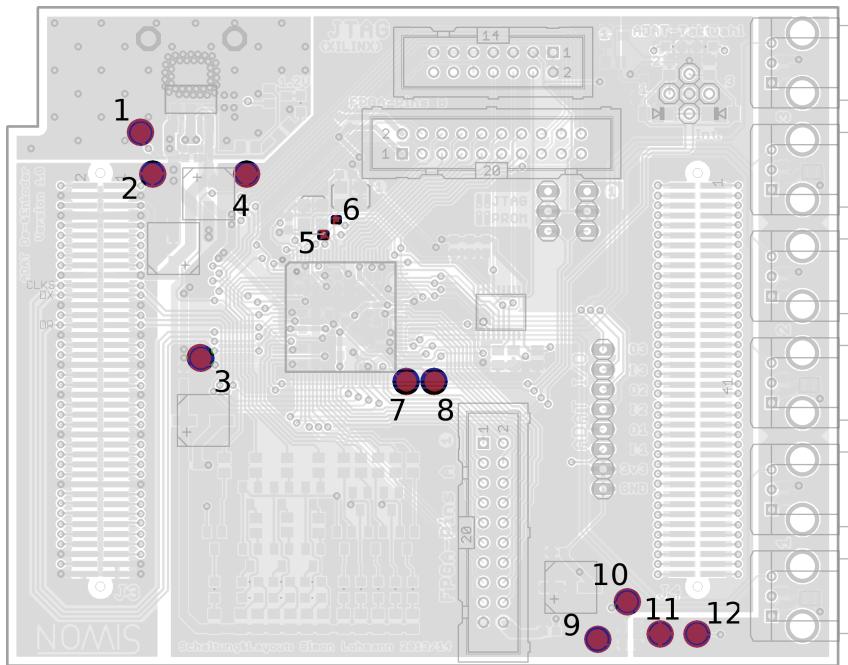


Abbildung E.12: Testpunkte auf der Platine

Nr.	Name	Beschreibung	Sollwert
1	TP_GEREGELELT_1V2	VCC <sub>INT</sub> am Spannungsregler	1,2 V
2	TP_EINGANG_GND	Masse am Spannungsregler	0 V
3	TP_FPGA_GND	Masse am FPGA	0 V
4	TP_EINGANG_3V3	3,3 V am Spannungsregler	3,3 V
5	TP_TAKT_ADAT	ADAT-Takt	12,288 MHz @ 3,3 V
6	TP_TAKT_50MHZ	Schneller Systemtakt	50 MHz @ 3,3 V
7	TP_FPGA_1V2	VCCINT am FPGA	1,2 V
8	TP_FPGA_3V3	3,3 V am FPGA	3,3 V
9	TP_ADAT_3V3	3,3 V vor der Analogfläche	3,3 V
10	TP_ADAT_GND	Masse vor der Analogfläche	0 V
11	TP_ADAT_3V3_ANALOG	3,3 V auf der Analogfläche	3,3 V
12	TP_ADAT_GND_ANALOG	Masse auf der Analogfläche	0 V

## E.5 Taktrouting im FPGA

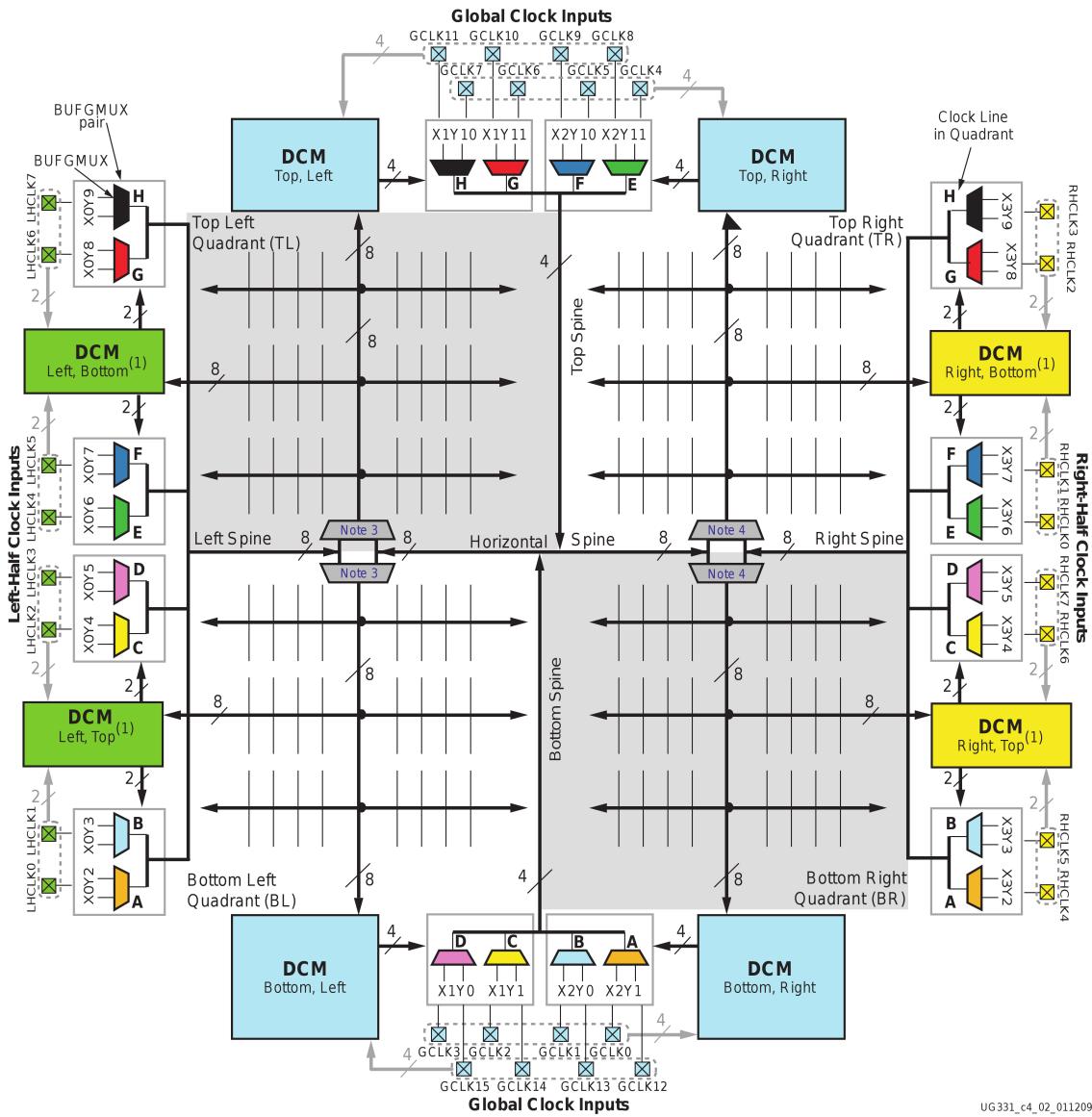


Abbildung E.13: Taktrouting im FPGA (Quelle: [Xil11, S. 47, Fig. 2-2])