Fachhochschule
Dortmund
University of Applied Sciences and Arts

Fachbereich Elektrotechnik

Bachelorarbeit

Entwicklung einer Ansteuerung eines an einem FPGA angeschlossenen DA-Wandlers unter Nutzung einer QT-Anwendung

Guellaf Othmane

Matrikelnummer: 7083351

Erstprüfer: Prof. Dr.-Ing. Michael Karagounis

Zweitprüfer: M.Eng. Andreas Stiller

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

DAC	Digital-Analog-Wandler
VHDL	**V**ery High-Speed Integrated Circuit **H**ardware **D**escription **L**anguage
FPGA	Field Programmable Gate Array
SPI	Serial Peripheral Interface
MSB	most significant bit

1 Einleitung

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist, die Entwicklung einer DA-Wandler-Ansteuerung unter Nutzung einer Qt-Anwendung.

Diese Arbeit ist eine Fortsetzung des im Rahmen der betrieblichen Praxis durchgeführten Projektes, in dessen Rahmen die Steuerung eines Digital-Analog-Wandlers in VHDL entworfen wurde.

Die beiden Arbeiten sind wiederum Bestandteil des POLDI-Projekts, welches zum Ziel hat mit Hilfe von polarisiertem Licht einen berührungslosen Winkelsensor zu entwickeln.

1.2 Einführung zum berührungslosen Winkelsensor(POLDI)

Das Projekt Polarisation Sensitive Photo Diodes (POLDI) dient der Entwicklung eines berührungslosen Winkelsensors, welcher mit Hilfe von polarisiertem Licht einen Winkel bestimmen soll. Die Lebensdauer von optischen Sensoren ist deutlich höher als von konventionellen Sensoren, da keine Reibung auftritt.

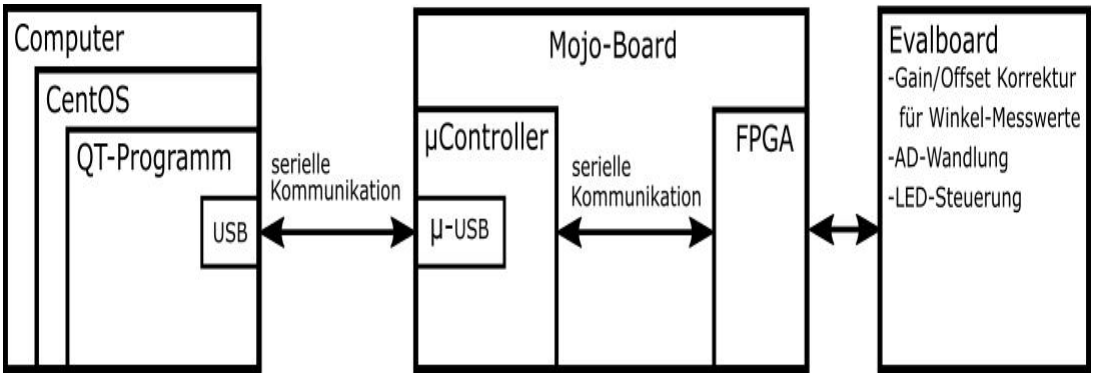

Abbildung 1: Datenaustausch zwischen Computer und POLDI

In Abbildung 1 ist der Aufbau des Systems schematisch abgebildet. Auf der linken Seite ist der Computer mit dem Betriebssystem CentOS und dem QT Programm dargestellt. Dieses verwendet den USB-Port zum Datenaustausch mit dem Mojo-Board, welches mit einem Mikrocontroller und einem FPGA bestückt ist und in der Mitte der Abbildung dargestellt ist. Rechts im Bild ist das EvalBoard erkennbar. Die Pfeile stellen den Datenverkehr dar, bei dem die Referenzspannung der ADCs und die Messwerte transferiert werden. (Duepertal, 2017)[1]

[1] (Duepertal, Projektarbeit 1, 2017)

1.3 Struktur der Arbeit

In Kapitel 1 wird ein kurzer Einblick in das Poldi-Projekt gegeben und der Datenaustausch zwischen Betriebssystem, Mojo-Board und Evalboard erläutert. Kapitel 2 gibt einen kurzen Einblick in die Hardwaresprache VHDL und erklärt die grundlegenden Eigenschaften der verwendeten Blockelemente.

Kapitel 3 listet die im Projekt verwendeten Werkzeuge und Software auf. Anschließend werden in Kapitel 4 die Funktionsweise von Digital-Analog-Wandlern anhand des im Rahmen dieser Arbeit verwendeten Bausteins DAC5686 beschrieben.

In Kapitel 5 werden die drei Blöcke, die für dieses Projekt von grosser Bedeutung sind (DAC 5686, AVR-Interface und Ansteuerung) erläutert und deren Simulationen beschrieben.

In Kapitel 6 werden die Ergebnisse dieser Arbeit unter Nutzung einer QT-Anwendung angegeben und theoretische Berechnungen mit praktischen Messungen verglichen. Anschließend wird im Kapitel 7 eine kurze Zusammenfassung gegeben.

Der VHDL der implementierten Module wird im Anhang offen gelegt.

2. Einführung in VHDL

VHDL (**V**ery High-Speed Integrated Circuit **H**ardware **D**escription **L**anguage) ist eine Hardwarebeschreibungssprache, die in der elektronischen Entwurfsautomatisierung verwendet wird, um digitale -Systeme zu entwerfen.

In der Computertechnik ist eine Hardwarebeschreibungssprache (**HDL**) eine spezialisierte Computersprache, die verwendet wird, um die Struktur und das Verhalten von elektronischen Schaltungen zu beschreiben.

Das allgemeine Format eines VHDL-Moduls basiert auf einer Struktur aus Blöcken, die sich zu komplexen Systemen zusammenstellen lassen.
Die grundlegenden Baueinheiten eines VHDL-Moduls erlaubt die einfache Beschreibung einer logischen Funktionsschaltung.

2.1 Blöcke eines VHDL-Programms

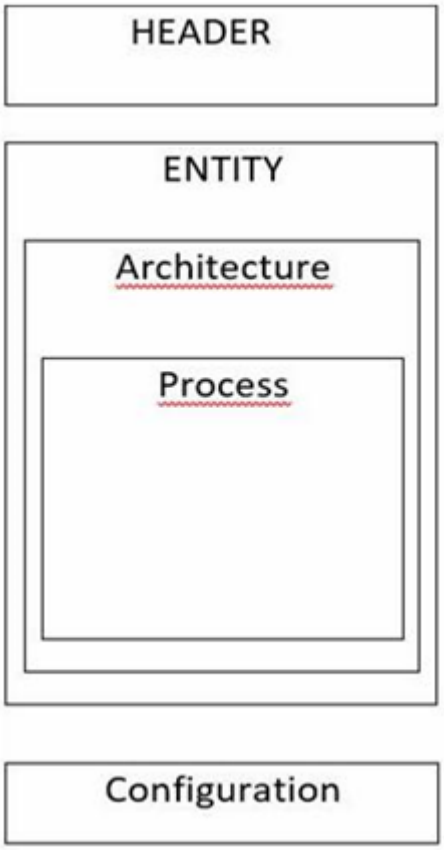

Abbildung 2: Struktur eines VHDL-Programms

2.1.1 Header

Im Header werden Definitionen bezüglich Bibliotheken und Pakete festgelegt, die für die folgende Schaltungsbeschreibung gelten sollen.

Bibliotheken (engl. Libraries) bieten eine Reihe von Paketen, Komponenten und Funktionen, die den Entwurf von Hardware vereinfachen.

Pakete (engl. Packages) bieten eine Sammlung von Datentypen und Funktionen bzw. Prozeduren.

Im Folgenden ist ein Beispiel für die Einbindung der **ieee**-Bibliothek und ihres Pakets std_logic_1164 angegeben.

Beispiel:

```
library ieee;
use ieee.std_logic 1164.all;
```

2.1.2 Entität

Die Entität (engl. Entity) ist die Beschreibung der Schnittstelle zwischen einem Design und seiner externen Umgebung. Sie kann auch Deklarationen und Anweisungen enthalten, die Teil der Design-Entität sind. Eine gegebene Entitätsdeklaration kann von vielen Designeinheiten gemeinsam genutzt werden, von denen jede eine andere Architektur besitzt. Somit kann eine Entitätsdeklaration möglicherweise eine Klasse von Design-Entitäten darstellen, die jeweils die gleiche Schnittstelle aufweisen.
Beispiel:

```
entity entity-name is
generic (generic_list);
port (port_list);
end entity_name;
```

2.1.3 Architektur

Eine Architekturanweisung definiert die Struktur oder Beschreibung eines Entwurfs und ist mit einer Entität verbunden, die vorher definiert wurde.

VHDL ermöglicht einer Entität, mehrere Architekturen zu besitzen. Die Architektur beschreibt das Verhalten der Entität. Das Verhalten kann auf verschiedene Arten beschrieben werden, zum Beispiel als Datenfluss-Modell, Struktur-Modell oder Verhaltens-Modell.

Die Architektur besteht aus zwei Teilen: nämlich dem Deklarationsteil und dem Anweisungsteil.

Der Deklarationsteil steht zwischen den Schlüsselwörtern **architecture** und **begin**. In diesem Abschnitt können Verbindungssignale, andere von dieser Architektur referenzierte Komponenten oder Konstanten definiert werden.

Der Anweisungsteil beginnt nach dem Schlüsselwort **begin** und beinhaltet die Anweisungen, Zuordnungen und Strukturen des Designs.

Beispiel:

```
architecture "architecture_name" of "entity_name" is
                declarations
//hier können lokale Signale deklariert werden
begin
                concurrent statements
//hier steht die Funktionsbeschreibung (nebenläufig)
end architecture;
```

2.1.4 Prozesse

Die Prozessanweisung repräsentiert das Verhalten eines Teils des Designs. Es besteht aus den sequentiellen Anweisungen, deren Ausführung in der vom Benutzer definierten Reihenfolge erfolgt.

Beispiel:

```
process
declarations;
 //hier können lokale Signale oder Variablen deklariert werden
begin
 sequential statements;
 //hier ist eine sequentielle Umgebung
 end process;
```

2.1.5 Konfiguration

Konfigurationen (engl. configurations) sind ein fortgeschrittenes Konzept in VHDL, welche sehr nützlich sein kann, wenn es die richtige Anwendung findet. Konfigurationen ermöglichen dem Designer, unterschiedliche Architekturen für eine einzelne Entität anzugeben. Dadurch können die Interna eines Designs geändert werden, während die Schnittstelle gleichbleibt. Konfigurationen sind nicht erforderlich, um ein grundlegendes VHDL-Design zu erhalten.

2.2 Behavioral-Stil Architektur

Es gibt drei verschiedene Modelle, um VHDL-Architekturen zu beschreiben. Diese sind bekannt als Datenfluss-Modell (engl. **data- flow**), Struktur-Modell (engl. **structural**) und Verhaltens-Modell (engl. **behavioral**).

Im Rahmen dieser Arbeit wurde hauptsächlich mit dem Verhaltens-Modell gearbeitet.

Verglichen mit dem Struktural-Ansatz gibt das Verhaltens-Modell keine Auskunft darüber, mit welchen Komponenten oder Gattern das Design in der verwendeten Technologie zu implementieren ist.

Der VHDL-Code, welcher in einem Verhaltens-Stil verfasst ist, modelliert lediglich wie Schaltungsausgänge gegenüber den Schaltungseingängen reagieren können. Im Struktural-Modell werden, wie bei einer Netzliste, Instanzen platziert und die Datenverbindungen zwischen ihnen definiert. Die Implementierung folgt dann der Beschreibung.

Behavioral ist eine Verhaltensbeschreibung und beschreibt das Verhalten der Schaltung abstrakt. Z.B. mit If-Abfragen und andere Sprachkonstrukten. Dies hat zur Folge, dass das Synthese Tool erst eine Schaltung daraus erzeugen muss, so, dass man keinen Einfluss über die Implementation hat.

Data-Flow ist eine Mischform: Hier wird zwar ebenfalls das Verhalten beschrieben, jedoch auf eine Weise welche die Struktur erkennbar macht.
Die Zeile c<=a xor b, beschreibt das Schaltverhalten, deutet aber gleichzeitig auch auf die Struktur der Schaltung und die Verwendung eines XOR-Gattern hin. wie schon an Hand dieses Beispiels deutlich wird findet in VHDL Modellen üblicherweise alle Beschreibungsformen gleichzeitig bzw. gemischte Anwendung.

3. Werkzeuge und Software

In diesem Kapitel werden die für die Implementierung der Schnittstelle verwendeten Software-Tools vorgestellt.

3.1 Xilinx Ise 14.7

Xilinx ISE (Integrated Synthesis Environment) ist ein Software-Werkzeug von Xilinx zur Synthese und Analyse von HDL-Designs, mit dem Entwickler ihre Designs synthetisieren (kompilieren), Timing-Analysen durchführen, die Reaktion eines Designs auf verschiedene Stimuli simulieren und den verwendeten Baustein konfigurieren können.

Xilinx ISE ist eine Designumgebung für FPGA-Produkte von Xilinx, und dementsprechend eng und die mit der Architektur solcher Chips ausgerichtet und kann nicht mit FPGA-Produkten anderer Anbieter verwendet werden.
In diesem Projekt wird Xilinx ISE hauptsächlich für die Synthese und den Entwurf von Schaltungen verwendet, während ModelSim für die Simulation verwendet wird. Weitere Komponenten, die mit der Xilinx ISE ausgeliefert werden, sind das Embedded Development Kit (EDK), ein Software Development Kit (SDK) und ChipScope Pro.

Seit 2012 wurde Xilinx ISE zugunsten der Vivado Design Suite eingestellt, welche die gleichen Funktionen wie ISE mit zusätzlichen Funktionen für die System-on-a-Chip- Entwicklung bietet. Xilinx hat die letzte Version von ISE im Oktober 2013 (Version 14.7) veröffentlicht und erklärt, dass ISE in die Nachhaltigkeitsphase seines Produktlebenszyklus eingetreten ist und es keine geplanten ISE-Releases mehr geben wird.

3.2 ISE Simulator

Mit dem ISE Simulator (ISim) lassen sich die verschiedenen Module in einem HDL Design simulieren. Da die Synthese-Dauer von der Größe des Designs abhängt, ist es vor allem bei sehr großen Designs wichtig, zuerst mögliche Fehler mittels Simulation auszuschließen. Zum Simulieren wird mittels

VHDL eine Testbench erstellt, die den Stimulus für das zu testende Design erzeugt.

3.3 Qt creator

Im Rahmen dieser Projektarbeit wird für die Erstellung von C-Programmen der Qt-Creator mit der Version "5.6.2" verwendet.

Qt Creator ist eine plattformübergreifende, integrierte Entwicklungsumgebung, die Teil des Qt-Frameworks ist und für die C ++ - Programmierung ausgelegt ist.

3.4 Mojo Loader

Der Mojo Loader wird benötigt, um das bin-File auf den Mojo zu übertragen und wird von Embedded Micro kostenlos zur Verfügung gestellt.

3.5 MOJO V3

Das Mojo V3 FPGA development board ist ein Kreditkarten-großes Entwicklungsboard, welches wegen seiner einfach gehaltenen Peripherie besonders für den Einstieg in die Hardware-implementierung mittels FPGAs geeignet ist. Das Mojo V3 besitzt 84 digitale IO Pins und 8 analoge Input Pins, auf die über drei Buchsenleisten zugegriffen werden kann.

Zusätzlich befinden sich auf dem Board 8 General Purpose LEDs sowie zwei Spannungsregler für einen Spannungsbereich von 4.8 V 12 V.Als FPGA ist der XC6SLX9 aus der Spartan 6 Familie im TQG144 Package verbaut worden, der durch einen ATmega32U4 und einen 4 Mbit onboard Flash Speicher konfiguriert wird.

Abbildung 3: Mojo V3 development board

Die MOJO Funktionen umfassen:

- 84digitale IO-Pins
- Spartan 6 XC6SLX9 FPGA
- 8 analoge Eingäng
- 8 Allzweck-LEDs
- 1 Reset-Taste
- 1 LED zeigt an, wenn das FPGA richtig konfiguriert ist
- On-Bord Spannungsregelung, die 4,8V - 12V verarbeiten kann
- Ein Mikrocontroller (ATmega32U4) zur Konfiguration des FPGA, der USB- Kommunikation und zum Auslesen der analogen Pins
- Arduino-kompatibler Bootloader, mit dem Sie auch den Mikrocontroller einfach programmieren können
- On-Board-Flash-Speicher zum Speichern der FPGA-Konfigurationsdatein

4. DA-Wandler(AD5686)

4.1 Funktionsweise eines Digital-Analog-Wandlers

Ein Digital-Analog-Wandler wandelt einen digitalen Wert in ein analoges Signal um, Im Englischen wird er als Digital Analog Converter (abgekürzt **DAC**) bezeichnet und ist somit das Gegenstück zu einem AD-Wandler.

Eine einfache Form eines DA-Wandlers findet sich nachfolgend in der Umsetzung eines sogenannten R2R-Netzwerks. Der dargestellte Wandler hat in diesem Beispiel eine Auflösung von 4 Bits:

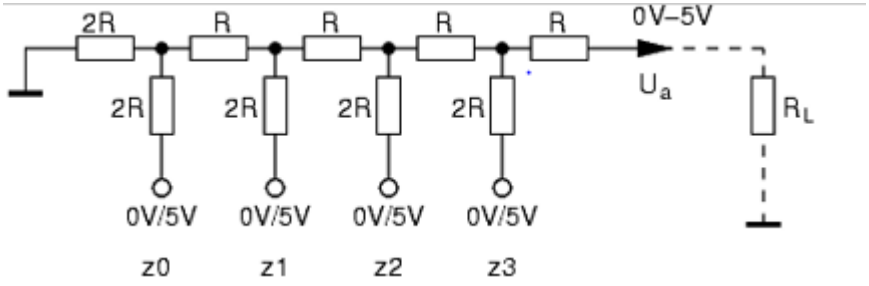

Abbildung 4: R2R-Netzwerk eines DAC-Umsetzers

Die Ausgangsspannung ist vom Lastwiderstand (RL) abhängig und berechnet sich nach der folgenden Formel:

$$U_a = \frac{U_{ref}}{2^4} \cdot \frac{R_L}{R + R_L} \cdot Z$$

Die wichtigsten Eigenschaften eines DA-Wandlers sind die Auflösung, die Genauigkeit und die Wandlungsgeschwindigkeit. Je höher die Auflösung, desto genauer kann ein DA-Wandler ei ne Spannung ausgeben. Die Auflösung wird in Bit angegeben und sagt aus, wie viele Abstufungen der DA Wandler für einen gegebenen Spannungsbereich generieren kann.

Beispiel:

besitzt ein DA-Wandler bispielweise 4 Bits, dann ist die Möglichkeit gegeben (2^4=16) verschiedene Spannungen auszugeben. kann er Signale in einem Spannungsbereich von bis zu MAX 5V verarbeiten, so beträgt die Spannungsdifferenz zwischen zwei gewandelten Werten mindestens:

5V/16 = 0, 3125 V = 312, 5 mV.

Im Prinzip gibt es bei den heutigen Präzisions-DACs zwei Architekturen: R2R und String. Bei beiden Architekturen handelt es sich um analoge Schaltkreise mit digitaler Steuerungslogik. Mit einer R2R-Architektur ist es möglich, entweder einen Ausgangsstrom oder eine Ausgangsspannung zu erzeugen. String-Architekturen können nur Ausgangsspannungen in kombination mit einem Ausgangsverstärker generieren.

4.2 Technische Rahmenbedingungen des AD5686

Der AD5686 ist ein 16-Bit-Digital-Analog-Wandler (VDAC) mit gepufferter Ausgangsspannung und einer Versorgungsspannung von 2,7 V bis 5,5 V, In Abbildung 6 ist das Funktionsblockdiagramm des AD5686 dargestellt.

Die serielle Datenschnittstelle bietet eine 3-Draht-Verbindung mittels SDIN, SCLK und SYNC als SPI- oder I2C-Interface.

Die serielle Schnittstelle des AD5686 arbeitet mit einer Frequenz von bis zu 50 MHz und enthält einen SDO-Pin, um es Benutzern zu ermöglichen, mehrere Geräte hintereinander zu verketten, was im englischsprachigen Raum als Daisy Chain bezeichnet wird.

Die 5686-Architektur besteht aus einem String-DAC gefolgt von einem Ausgangsverstärker.

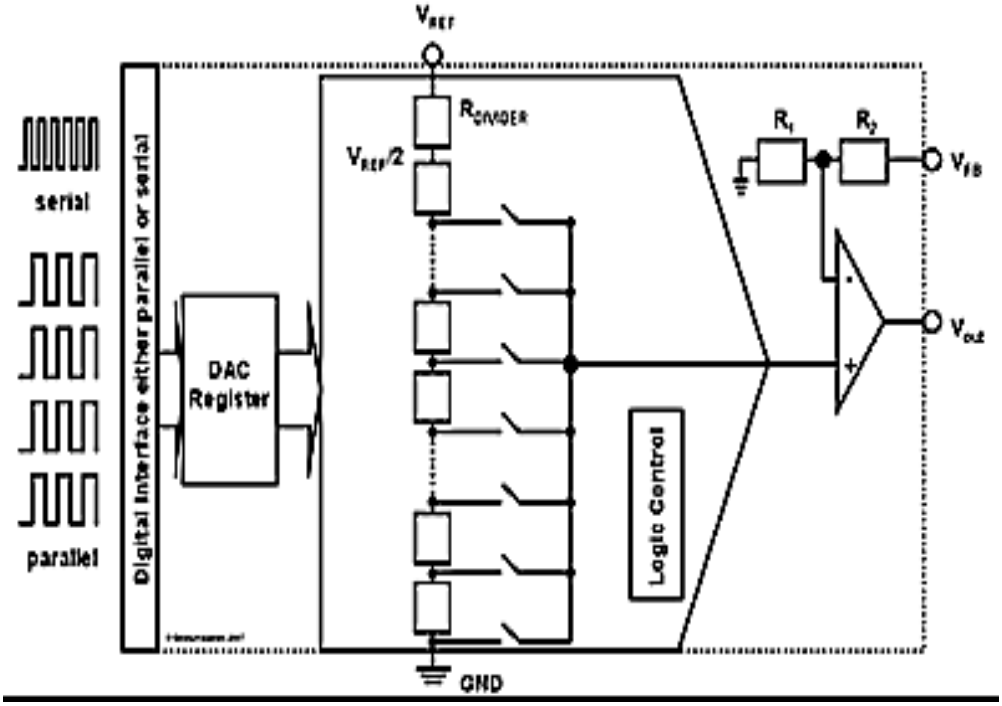

Abbildung 5: String-Architektur mit Spannungsausgang

Wie die Bezeichnung String-Architektur vermuten lässt, bilden bei dieser Architektur in Serie geschaltete Widerstände einen String (Reihe von Widerständen in Serie). Theoretisch braucht man 256 (2^8 = 256) Widerstände um einen 8-Bit-DAC zu implementieren

4.2.1 Blockdiagramm des AD 5686

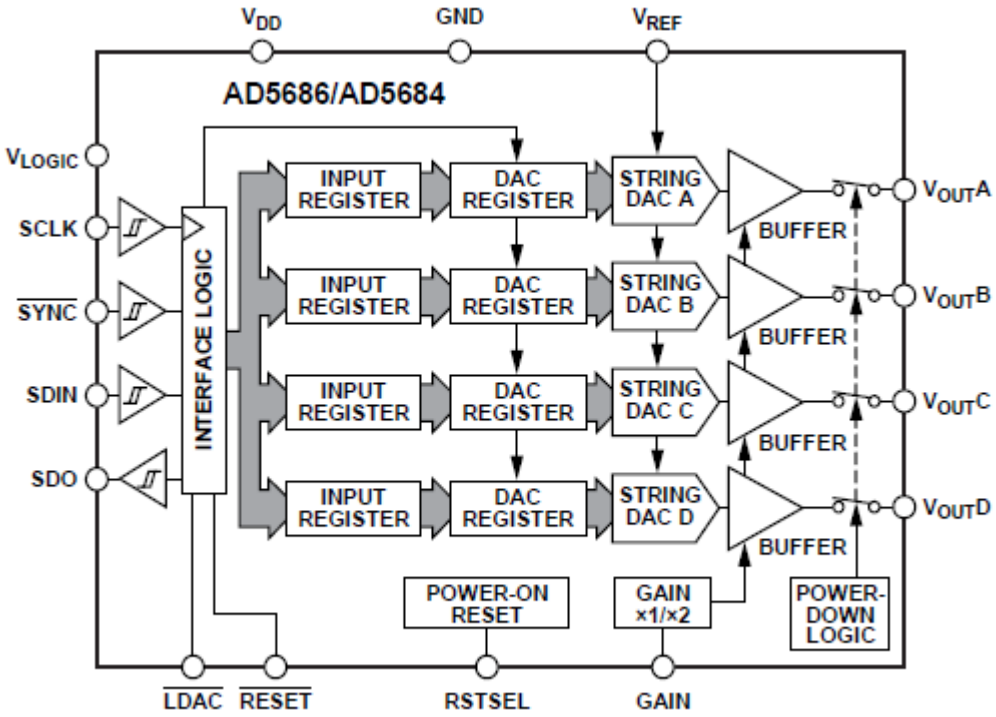

Abbildung 6: Funktionales Blockdiagramm des DAC 5686 (Devices, 2012-2017)

4.2.2 Serielle Datenübertragung

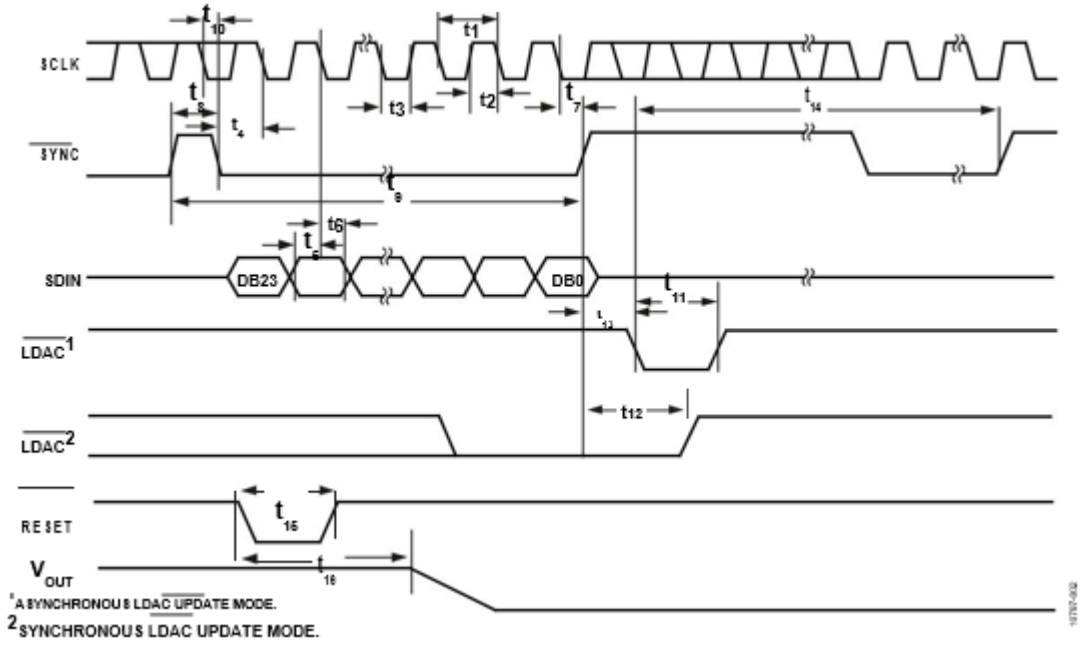

Abbildung 7: :Serielle Schreiboperationen des AD5686 (Devices, 2012-2017)[2]

[2] (Devices, Analog Devices, 2012-2017)

SCLK (Serial Clock Input):

Die Daten werden bei der fallenden Flanke des seriellen Takteingangs in das Eingangsschieberegister (engl. Input Shift Register) geschrieben und können mit einer Frequenz von bis zu 50 MHz übertragen werden.

SDIN (Serial Data Input):

Der **SDIN** Pin ist der Eingang des 24-Bit-Eingangsschieberegisters Daten werden bei der fallenden Flanke des seriellen Takteingangs (**SCLK**) in das Register übertragen.

SYNC (Active Low Control Input)

Wird **SYNC** Low gesetzt, werden Daten mit den fallenden Flanken der nächsten 24 Takte (**SCLK**) übertragen.

LDAC

Nimmt der **LDAC** Pin den Pegel Low an, wird jedes oder alle DAC-Register aktualisiert, wenn die Eingangsregister neue Daten erhalten haben. Dadurch können alle DAC-Ausgänge gleichzeitig aktualisiert werden, dieser Pin kann dauerhaft Low bleiben.

RESET

Der Baustein reagiert auf eine fallende Flanke, des am Reset Pin anliegenden Signals. Wird dieser Pin auf Low gesetzt, werden alle **LDAC**- Impulse ignoriert.

SDO (Serial Data Output):

Der **SDO**-Pin kann verwendet werden, um eine beliebige Anzahl von AD5686 / AD5684-Bausteinen in Reihe zu verketten oder um die empfangenen Daten wieder zurück zu lesen. Die seriellen Daten werden bei der steigenden Flanke von SCLK übertragen und sind bei der fallenden Flanke des Taktes gültig.

Der AD5686 hat ein 24-Bit-Eingangsschieberegister welches mit dem MSB zuerst geladen wird. Die ersten vier Bits sind die Befehlsbits (engl. Command-Bits) C3 bis C0, gefolgt von 4-Bit-DAC-Adressbits DAC A, DAC B, DAC C, und DAC D), gefolgt von 16-Bit-Eingangsdatenbits (Abbildung 8). Diese 16 Datenbits werden bei der fallenden Flanke von **SCLK** an das 24-Bit-Eingangs-register übertragen und bei der steigenden Flanke von **SYNC** aktualisiert. Die Definition der Befehlsbits ist Abbildung 10 zu entnehmen.

Abbildung 9 beschreibt die Kombination der Adressbits und der durch die entsprechende Bitkombination ausgewählten DACs.

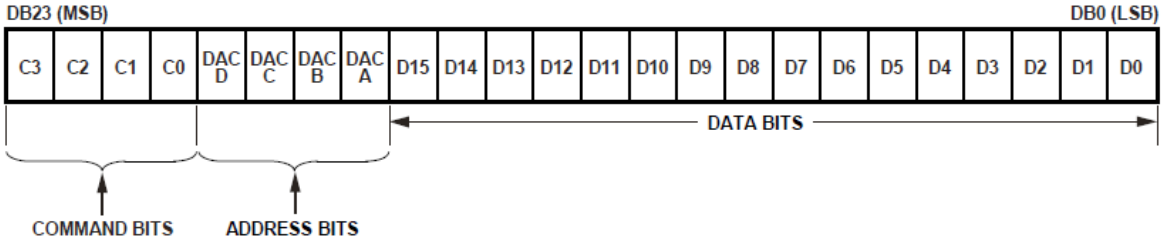

Abbildung 8: Eingangsschieberegister des DAC 5686 (Devices, 2012-2017)[3]

Address Bits				
DAC D	**DAC C**	**DAC B**	**DAC A**	**Selected DAC Channel**[1]
0	0	0	1	DAC A
0	0	1	0	DAC B
0	1	0	0	DAC C
1	0	0	0	DAC D
0	0	1	1	DAC A and DAC B
1	1	1	1	All DACs

Abbildung 9: Commandbits Definitionen des DAC 5686 (Devices, 2012-2017)

[3] (Devices, Analog Devices, 2012-2017)

4.2.3 Schreib-und Aktualisierungsbefehle

Command Bits				
C3	C2	C1	C0	Description
0	0	0	0	No operation
0	0	0	1	Write to Input Register n (dependent on $\overline{LDAC}$)
0	0	1	0	Update DAC Register n with contents of Input Register n
0	0	1	1	Write to and update DAC Channel n
0	1	0	0	Power down/power up DAC
0	1	0	1	Hardware LDAC mask register
0	1	1	0	Software reset (power-on reset)
0	1	1	1	Reserved
1	0	0	0	Setup DCEN register (daisy-chain enable)
1	0	0	1	Setup readback register (readback enable)
1	0	1	0	Reserved
...	...	...	...	Reserved
1	1	1	1	No operation, daisy-chain mode

Abbildung 10: Address Bits und ausgewählte DACs (Devices, 2012-2017)[4]

- Command 0001 (engl. Write to Input Register n)

Befehl 0001 erlaubt dem Benutzer, an jedes einzelne DAC-Input-register zu schreiben.

- Command 0010 (engl. Update DAC Register n with Contents of Input Register n)

Dieser Befehl lädt die DAC-Register / -Ausgänge mit dem Inhalt des ausgewählten Eingangsregisters und aktualisiert die DAC-Ausgänge direkt.

- Command 0011(engl. Write to and Update DAC Channel n)

Mit dem Befehl 0011 kann der Benutzer in die DAC-Register schreiben und die DAC-Ausgänge direkt aktualisieren.

[4] (Devices, Analog Devices, 2012-2017)

5 VHDL-Code und Simulation

5.1 DAC 5686

In diesem Kapitel wird das entworfene VHDL vorgestellt und die Simulationsergebnisse beschrieben.

5.1.1.Block des Moduls DAC5686

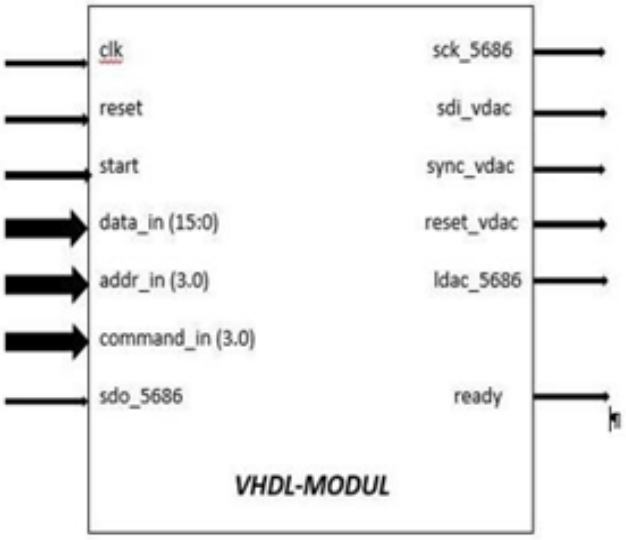

Abbildung 11: : Block des VHDL-Moduls(DAC5686)

5.1.2 Ablauf der Steuerung

Das VHDL-Modul wurde mit Hilfe einer State-Machine entworfen.
Hier sind die Zustände der State Machine im Einzelnen erklärt:

Zustand 0 (**state0**):
Hier wird zuerst gefragt, ob **start** auf 1 gesetzt ist. Ist dies der Fall, werden Anfangswerte für die Variablen **data_in_var**, **ready_var**, **state**, **sync_vdac_var**, **sdi_vdac_var** und **sck_5686_var** gesetzt. Bei einer steigenden Flanke des **CLK** Signals wird die Variable **sync_vdac_var** auf 1 gesetzt.

- Zustand 1(**state1**):

Die Variable sync_vdac_var wird wieder auf 0 gesetzt. Dann werden bei der Nächsten steigenden Flanke (clk) die Daten (unsigned(**command_in**)&unsigned(**addr_in**)& unsigned(**data_in**)) beginnend mit dem MSB in die Variable sdi_vdac_var geschrieben.

- Zustand 2 (**state2**):

Hier wird mit der Variable **counter** (Zähler) gearbeitet. dieser Zähler zählt von 0 bis 23 und fängt im Zustand 1 bei 23 an (MSB). es wird immer abgefragt, ob der Zähler ungleich null ist. Wenn ja, wird der Zähler um 1 dekrementiert, und es wird in den Zustand 1 zurück gewechselt.
Im Zustand 1 wird das nächste Bit (LSB) in die Variable **sdi_vdac_var** geschrieben**.** wenn der Zähler die Null erreicht und alle Daten geschrieben, sind. wird in den nächsten Zustand **(state 3)** gesprungen

- Zustand 3 (state3):

Hier wird bei einer steigenden Flanke die Variable **sync_vdac_var** wieder auf 1 gesetzt, und der Automat wechselt in den nächsten Zustand (10).

- Zustand 10 (state 10):

Hier wird, falls **start** Null ist, die Variable **ready_var** auf 1 gesetzt, und die state Machine beginnt wieder im Zustand 0.

In der Abbildung 12 wurde der innere Ablauf des VHDL-Moduls mithilfe eines Zustandsübergangsdiagramms beschrieben.

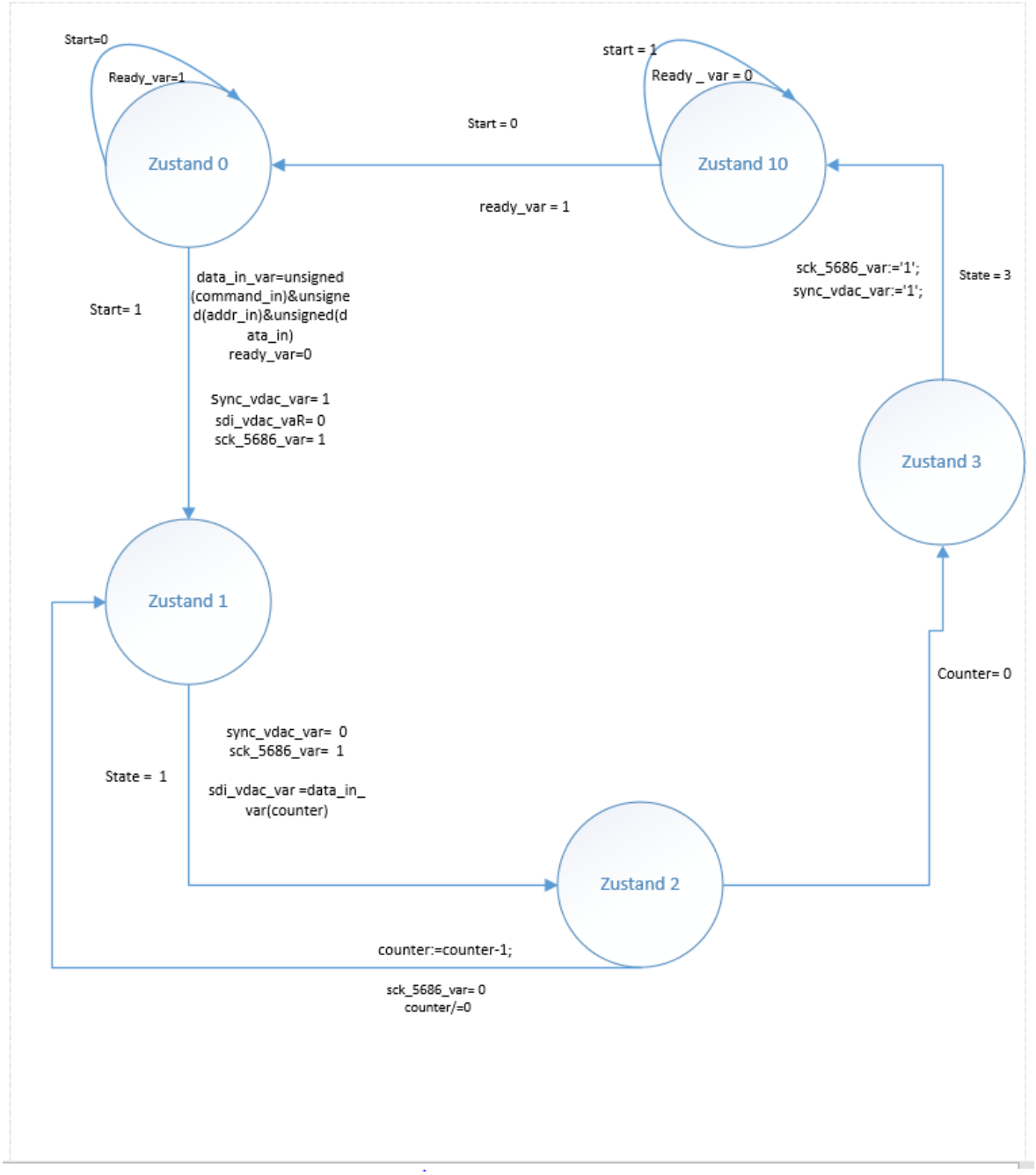

Abbildung 12: Zustandsübergangsdiagramm des VHDL-Moduls (DAC5686)

5.1.3 Die Simulation

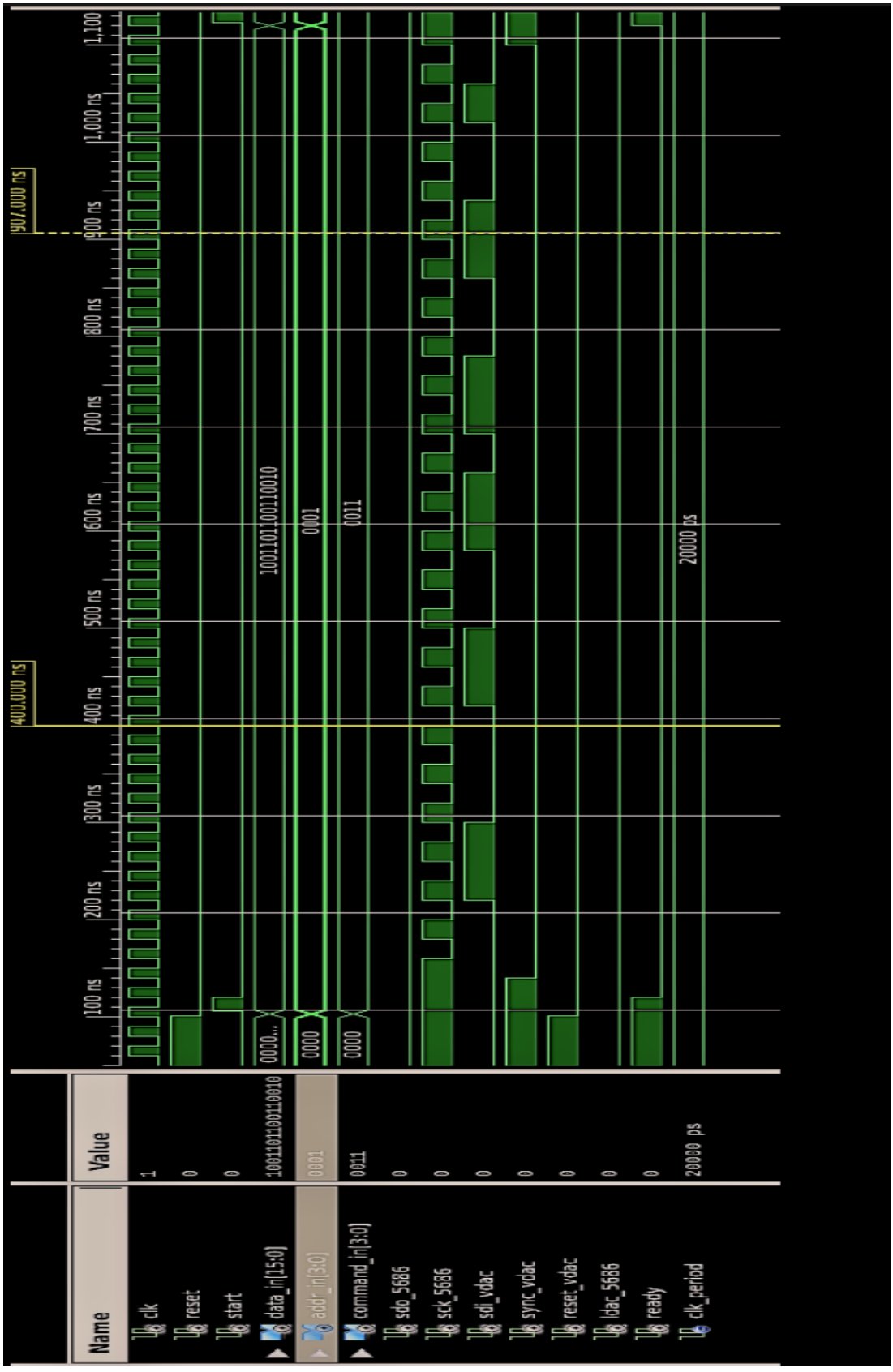

Abbildung 13: Simulation des Moduls DAC5686

Abbildung 13 zeigt das Ergebnis der Simulation des VHDL-Codes mittels einer Testbench. Das Signal **clk** (Clock) steht für den Grundtakt. Der Takt besitzt eine Periodendauer von 20 Nanosekunden. Nach 100 Nanosekunden wird **Reset** auf 0 und gleichzeitig **Start** auf 1 gesetzt. Dann wird abgefragt, ob **ready** high. Wenn ja erfolgt nach einer Wartezeit von 3 Nanosekunden die Datenübertragung in **sdi_vdac.**

Bei einer fallenden Flanke des Signals **sck_5685** folgen zunächst die Commandbits. Anschließend erscheinen die Bits für Adress und Data. In **command_in [3:0]** wird das Kommando 0011 gesetzt, da dieses Kommando den ausgewählten DAC-Kanal beschreibt und aktualisiert. Das Signal **addr_in [3:0]** wurde auf die Adresse 0001 des DAC Kanals A gesetzt der DAC A mit dem Befehl 0001 ausgewählt, da bei diesem Test nur ein DAC benötigt wird. Das Signal **data_in [15:0]** ist das zu übertragende Datenwort und kann für die Simulation beliebig gewählt werden.

Die anderen Signale, welche im Folgenden beschrieben werden, sind für die Steuerung des DACs verantwortlich.
Das Signal **reset_vdac** (Ausgang) folgt dem Signal **reset** (Eingang) und wird low, wenn das Signal **reset** wieder auf 0 gesetzt ist. Das Signal **ldac_5686** bleibt, wie vorher erwähnt, dauerhaft low.

Das Signal **sync_vdac** wird vor der Datenübertragung 0, nachdem auch das Signal **ready** auf 0 geht, und wird am Ende der Datenübertragung wieder 1, bevor das Signal **ready** wieder auf 1 geht.

Im Abbildung 12 sieht man, dass die Datenübertragung fehlerfrei abgelaufen ist und alle 24 Bits in die Variable **sdi_vdac** geschrieben worden sind. Mit Hilfe der Testbench ist damit sichergestellt worden, dass der Block die geforderte Funktionalität fehlerfrei abbildet.

5.2 AVR-Interface

Das Modul avr_interface wurde vom Hersteller des Mojo Boards mitgeliefert. Dieses Modul verwaltet die Kommunikation mit dem AVR-Mikrocontroller auf der Mojo-Platine.
Alle für das Projekt verwendete Module sind am Ende dieses Dokumentes angehängt.

5.3 Die Ansteuerung

Dieses Kapitel beschreibt das Modul Ansteuerung, welches das DAC5686 Modul mit dem Modul avr_interface verbindet

5.3.1 Erläuterung des Moduls

Wie in Abbildung 13 dargestellt reagiert das Modul Ansteuerung auf das Signal **new_rx_data** und übernimmt Byte-Weise die über die USB Schnittstelle empfangenen Daten. Nachdem alle für die Ansteuerung des DACs notwendigen Daten empfangen und formatiert worden sind, werden diese an das Modul DAC5686 übergeben.

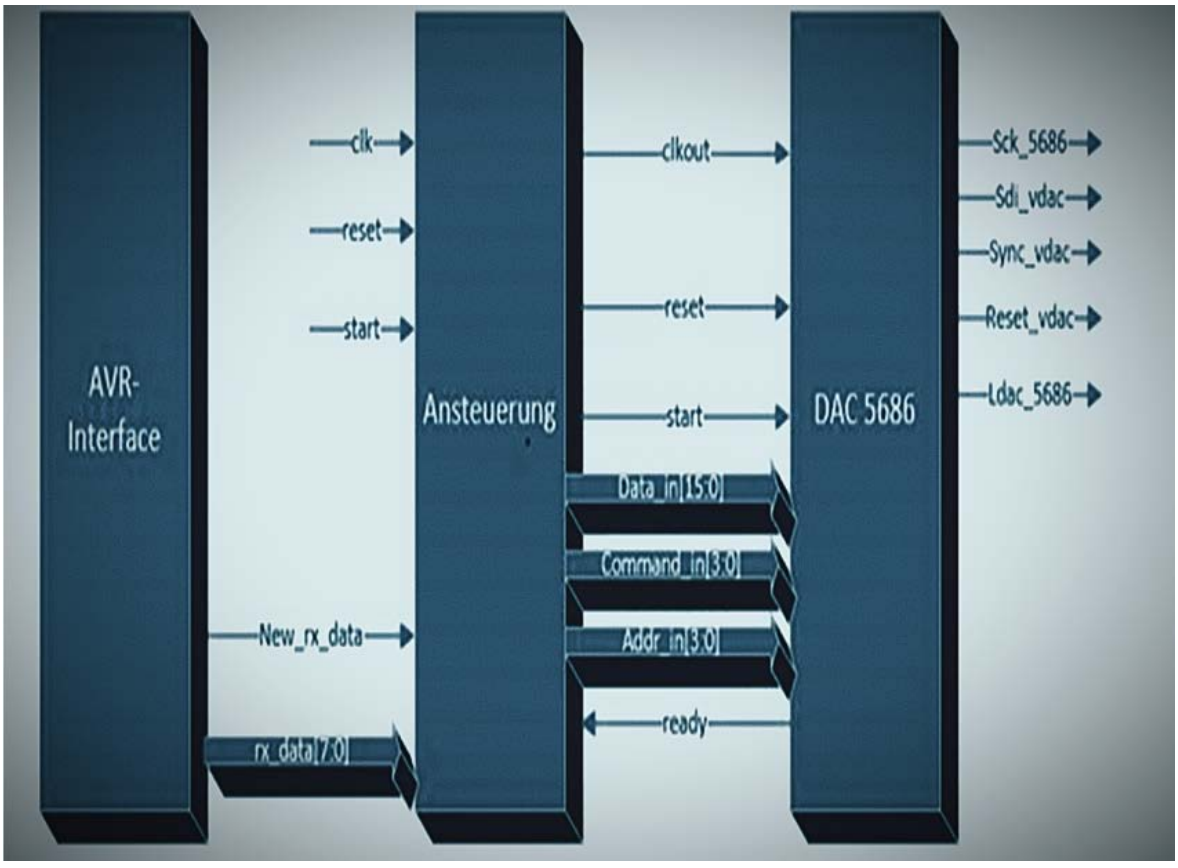

Abbildung 14: Block des Ansteuerungs-Moduls

Der VHDL Quell-Code des Moduls (DAC_Ansteuerung_top.vhd) ist am Ende dieses Dokuments angehängt.

5.3.2 Simulation des Moduls

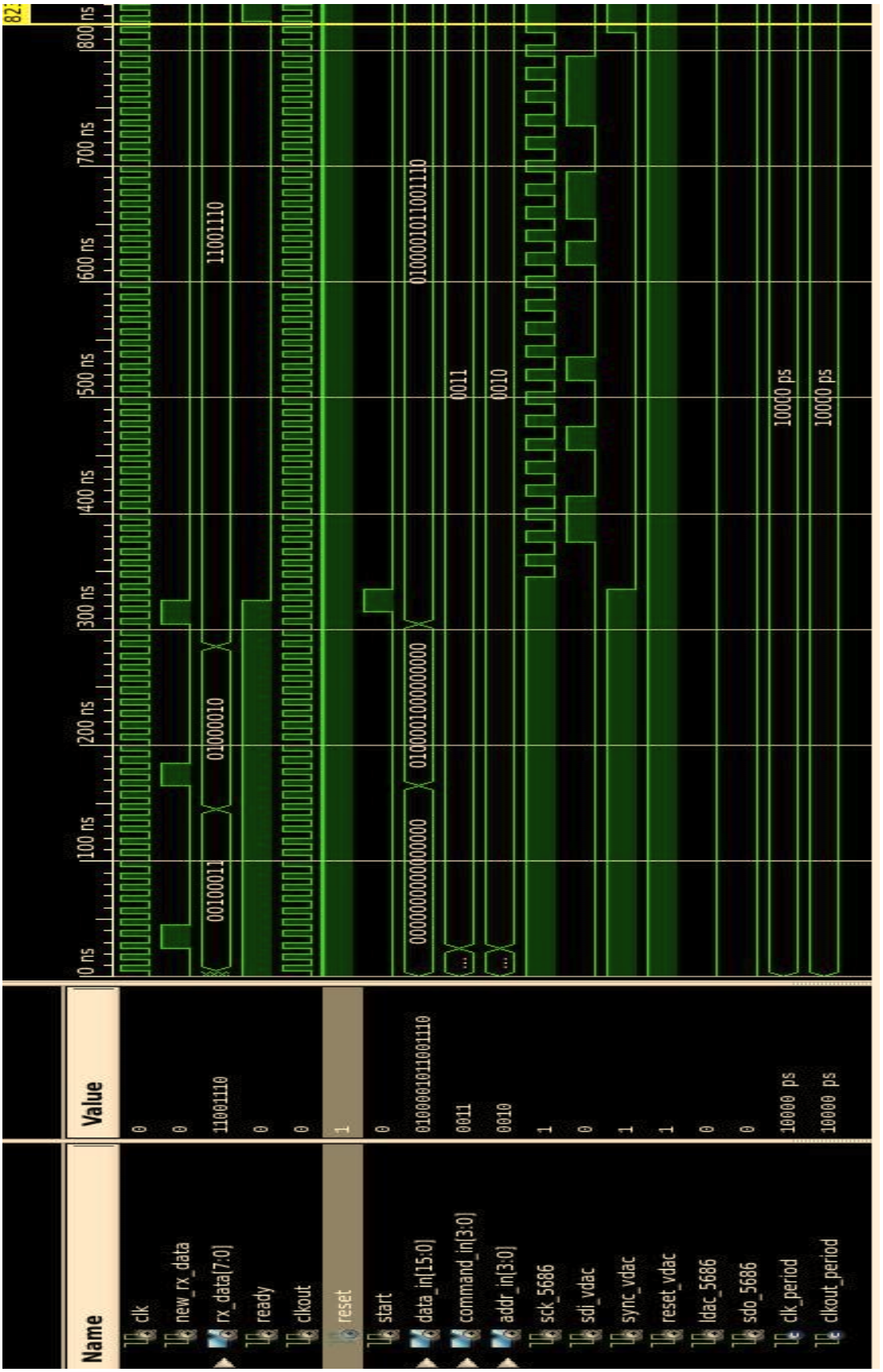

Abbildung 15: Simulation des Ansteuerungs-Moduls mittels Testbench

Die Grafik zeigt das Ergebnis der Simulation des Ansteurungsmoduls mittels einer Testbench. Die **clk** (Clock) steht für den Grundtakt. Diese arbeitet mit einem Takt von 10 Nanosekunden.

Nach 20 Nanosekunden wird das Signal **new-rx-data** auf 1 gesetzt und gleichzeitig werden die ersten acht bits (command und addr) in **rx-data** geschrieben. Dieser Vorgang wird mit einem Intervall von 100 ns noch zweimal wiederholt bis die anderen 16 bits (data) auch in **rx-data** geschrieben worden sind.

Wenn dieser Vorgang abgeschlossen ist, geht das Signal **ready** auf 0 und ebenso das Signal **sync_vdac**.

Direkt im Anschluss wird das Signal **start** auf 1 gesetzt, und gleichzeitig wird in den nächsten 24 Takten des Signals **sck_5686** die Daten (24 Bits) seriell auf dem Bus **sdi_vdac** übertragen.

Wenn alle 24 Bits über sdi_**vdac** übertragen worden sind, geht das Signal **ready** wieder auf 1 und ebenfalls das Signal **sync_vdac.**

6. Ergebnisse der Qt-Anwendung

6.1 Die QT-Anwendung

Für die Dateneingabe wird die Widget Anwendung aus der Arbeit [Dueppert-hal,15] verwendet.

In Abbildung15 ist die grafische Bedienoborfläche dargestellt. Zu sehen sind die beiden Textfelder zur Ein- und Ausgabe der transferierten Daten wobei das linke Feld zur Eingabe von Charactern dient. Darüberhinaus ist auch der Button zum Senden von Datensätzen zu sehen.

Abbildung 16: Qt-Anwendung (Duepertal, 2017)[5]

Das Qt-Programm wurde in dieser Arbeit geändert und ergänzt, damit anstelle von drei ASCII-Zeichen, sechs hexadezimale Zeichen eingegeben werden können.

[5] (Duepertal, Projektarbeit 1, 2017)

6.2.Die DAC-Übertragungsfunktion

Die Übertragungsfunktion des Digital-Analog-Wandlers ist folgendermaßen beschrieben:

$$V_{OUT} = V_{REF} \times Gain\left[\frac{D}{2^N}\right]$$

Dabei gilt:

- N: ist die Auflösung des Digital-Analog-Wandlers.
- D: ist das dezimale Äquivalent des Binärcodes, der in□das DAC-Register geladen wird und kann einen Wert zwischen 0 und (2hoch16) = 65536 annehmen
- Gain: ist die Verstärkung des Ausgangsverstärkers und ist standardmäßig auf 1 eingestellt.

  Gain kann durch das Gain-Select-Pin auf ×1 oder ×2 eingestellt werden.

  Wird der GND-Pin an GND angeschlossen ist, haben alle DAC-Ausgänge einen Spannungsbereich von 0V bis Vref.

  Wird der Gain Pin an VDD angeschlossen, haben alle DAC-Ausgänge einen Spannungsbereich von 0V bis 2 x Vref.

  In unserem Projekt beträgt die Verstärkung dem Faktor Eins (×1), da das Gain-Select-Pin des AD 5686 Bausteins an GND angeschlossen ist, wie dem Layout der Poldi Platine entnommen werden kann.

- Vref: Referenzspannung, in unserem Projekt beträgt diese 3V.

6.3 Messergebnisse

Je nachdem welche sechsstelligen Hexadezimalzahl eingegeben wird, können die DAC-Ausgänge theoretisch eine Ausgangsspannung zwischen 0 und 2,99995 V ausgeben.

Diese Spannungen werden in hexadezimaler Darstellung durch die Daten sind in Bereich von 310000 bis 31FFFF erreicht.
Die ersten zwei Ziffern (31(hex)= 00110001(binär)) müssen dabei immer gleich gewählt werden, stellen dabei die Felder Command und Addr dar und werden unabhängig vom gewünschten Spannungswert immer gleich gewählt, das richtige Kommando abzusetzen und denselben DAC Kanal anzusteuern. gedacht sind.

Der erste Befehl 0011(binär) erlaubt dem Benutzer, an jedes DAC-Register zu schreiben und den DAC-Ausgang direkt zu aktualisieren.

Der zweite Befehl 0001(binär) erlaubt dem Benutzer, den DAC-Ausgang A zu schreiben.

Die letzten Vier Ziffern 0000(hex) bis 1111(hex) entsprechen den DAC Daten. Mit dieser Bitkombination stellt man ein, welche Spannung der DAC-Ausgang annehmen soll.

6.3.1 Theoretische Berechnungen

- Die kleinste Spannung ergibt sich mit dem Bitmuster 310000(hex).
  Vout = 3V*(0/65536) = 0V 0000(hex)=0(Dec)
- Die höchste Spannung ergibt sich mit dem Bitmuster 31FFFF(hex).
  Vout = 3V*(65535/65536) =2,99995 V FFFF(hex)=65535(Dec)
- Der mittlere Wert der Ausgangsspannung ergibt sich mit dem Bitmuster 318000(hex).
  Vout=3V*(32768/65536) = 1,5 V 8000(Hex)=32768(Dec)

6.3.2 Praktische Messungen

In diesem Kapitel wird die Durchführung der Messung dokumentiert. Es wurden die im vorherigen Abschnitt berechnet Bittmuster in die QT Oberfläche eingegeben und über USB an den Mojo und das Poldi Board übertragen. Die DAC Ausgangsspannung wurde mit eime Voltmeter gemessen.

Der kleinste Wert der Spannung:

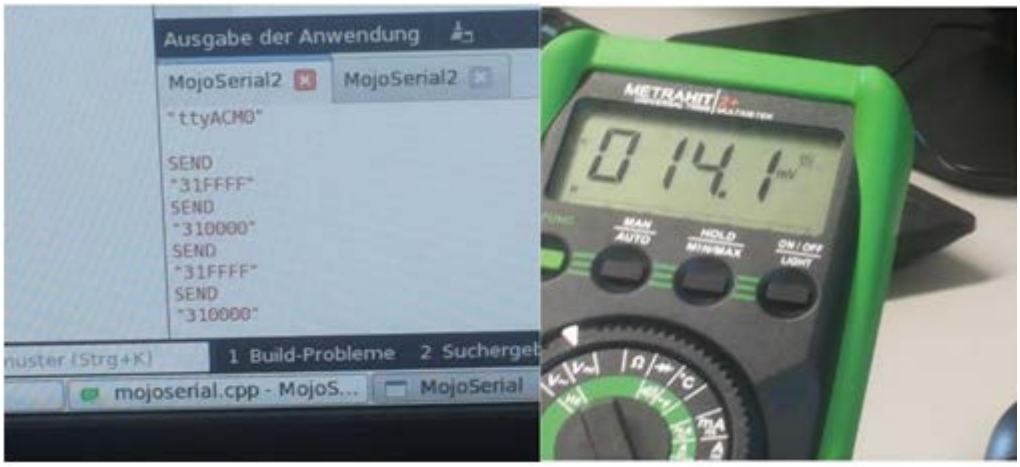

Abbildung 17: Messung der kleinste Wert der Spannung

Der höchste Wert der Spannung:

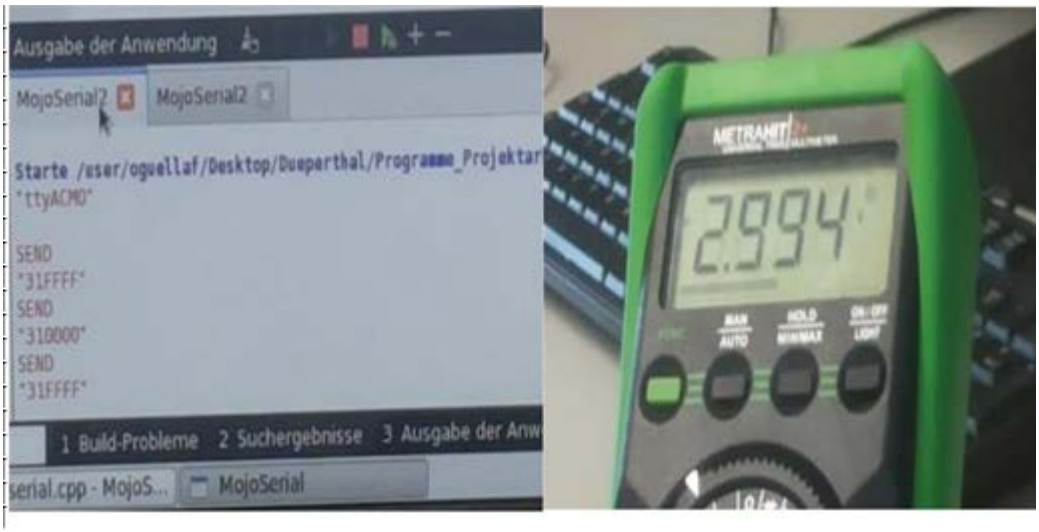

Abbildung 18: Messung der höchste Spannung

Messung der mittlere Wert der Spannung:

Gemessen wurde bei Eingabe von 318000(bzw118000) eine Spannung von 1,495 V.

6.3.3 Messabweichung

Die Messungen haben im Vergleich zum Theoretische Berechnungen einen Messabweichung von nicht mehr als -1%, was den Angaben im Datenblatt des AD 5686-Bausteins entspricht.

Diese wurde wie folgendes berechnet:

((Wm-Wt) /Wm) * 100

Dabei gilt:

Wm: gemessener Wert.

Wt: der, an Hand der Übertragungsfunktion berechnete Wert.

In folgender Tabelle wird ein Vergleich zwischen theoretische Berechnungen und praktische Messungen für bestimmte Bitmuster durchgeführt.

.

Bitmuster	Dezimale äquivalent	Theor. Berechn.	Prakt. Messung	Mess. Abweich.
316262	25186	1,153V	1,149 V	- 0,34%
31BE91	48785	2,233 V	2,229 V	- 0,18%
31FFFF	65535	2,999 V	2,994 V	- 0,17%
318000	32768	1,5 V	1,495 V	- 0,33%
312121	8481	0,388 V	0,385 V	- 0,11%

7. Zusammenfassung und Ausblick

In diesem Projekt ging es um die Entwicklung einer Ansteuerung eines an einem FPGA angeschlossenen DA-Wandlers unter Nutzung einer QT-Anwendung.
Dazu gehörte auch der Entwurf eines Ansteuerungsmoduls, das die beiden Module DAC5686 und AVR-Interface verbindet. Die Funktion dieses Moduls (Ansteuerung) wurde durch eine Simulation geprüft und ergab eine einwandfreie Funktion.
Das Qt-Programm, welches aus einer anderen Arbeit übernommen worden ist, wurde geändert und ergänzt, so dass statt ASCII-Zeichen hexadezimale Zahlen eingegeben werden können, um den kompletten DAC Wertebereich konfigurieren zu könne.
Mit der entwickelten QT-Anwendung kann die Funktionsweise der entworfenen Ansteuerung des DAC geprüft und bestätigt werden, indem die theoretischen Rechnungen mit den gemessenen Werten verglichen werden.
Der Vergleich zeigte, dass diese übereinstimmen mit einer kleinen Abweichung von wenigen Millivolts.
Die Resultate dieser Projektarbeit sollen im Projekt POLDI (Entwicklung eines optischen Winkelsensors) verwendet werden.

Literaturverzeichnis

Devices, A. (2012-2017). *Analog Devices.*

Devices, A. (2012-2017). *Data sheet.*

Duepertal. (2017). *Projektarbeit 1*. Dortmund.

Anhänge

Anhang A: AVR Interface

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity avr_interface is
	port (
		clk: in std_logic;
		rst: in std_logic;
		cclk: in std_logic;
		rx: in std_logic;
		rx_data: out std_logic_vector(7 downto 0);
		new_rx_data: out std_logic
	);
end avr_interface;

architecture RTL of avr_interface is

		signal ready: std_logic;
		signal n_rdy: std_logic;
begin

n_rdy <= NOT ready;

cclk_detector : entity work.cclk_detector
port map (
		clk	=> clk,
		rst	=> rst,
		cclk	=>	cclk,
		ready	=>	ready
);


serial_rx	: entity work.serial_rx
generic map (
CLK_PER_BIT	=> 100,
CTR_SIZE	=> 7
)

port map (
		clk	=> clk,
		rst	=> n_rdy,
		rx	=> rx,
		data	=> rx_data,
		new_data	=> new_rx_data
);

end RTL;
```

Anhang B: DAC- Ansteuerung _TOP

```
entity DAC_Ansteuerung_top is
      Port (
                  rx: IN std_logic;
                  clk: IN std_logic;
                  rst: IN std_logic;
                  cclk: IN std_logic;
                  sck_5686 : OUT  std_logic;
                  sdi_vdac : OUT  std_logic;
                  sync_vdac : OUT  std_logic;
                  reset_vdac : OUT  std_logic;
                  ldac_5686 : OUT  std_logic;
                  sdo_5686 :IN std_logic;
                  new_rx_data_mess:out std_logic
                   );
end DAC_Ansteuerung_top;

architecture Behavioral of DAC_Ansteuerung_top is

      COMPONENT Ansteurung
    PORT(
         clk : IN  std_logic;
         new_rx_data : IN  std_logic;
         rx_data : IN  std_logic_vector(7 downto 0);
         ready : IN  std_logic;
         clkout : OUT  std_logic;
         reset : OUT  std_logic;
         start : OUT  std_logic;
         data_in : OUT  std_logic_vector(15 downto 0);
         command_in : OUT  std_logic_vector(3 downto 0);
         addr_in : OUT  std_logic_vector(3 downto 0)
        );
    END COMPONENT;

       COMPONENT DAC5686
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         start : IN  std_logic;
         data_in : IN  std_logic_vector(15 downto 0);
         addr_in : IN  std_logic_vector(3 downto 0);
         command_in : IN  std_logic_vector(3 downto 0);
         sdo_5686 : IN  std_logic;
         sck_5686 : OUT  std_logic;
         sdi_vdac : OUT  std_logic;
         sync_vdac : OUT  std_logic;
         reset_vdac : OUT  std_logic;
         ldac_5686 : OUT  std_logic;
         ready : OUT  std_logic

        );
    END COMPONENT;

       COMPONENT avr_interface

      PORT (
            clk: in std_logic;
            rst: in std_logic;
            cclk: in std_logic;
            rx: in std_logic;
            rx_data: out std_logic_vector(7 downto 0);
            new_rx_data: out std_logic );

       END COMPONENT ;
```

```
        signal new_rx_data : std_logic ;
        signal rx_data : std_logic_vector(7 downto 0) ;
        signal ready : std_logic;
        signal reset : std_logic;
        signal start : std_logic;
        signal data_in : std_logic_vector(15 downto 0);
        signal addr_in : std_logic_vector(3 downto 0);
        signal command_in : std_logic_vector(3 downto 0);
        signal clkout:std_logic;
        signal n_rst:std_logic;

begin
       new_rx_data_mess<=new_rx_data;
       n_rst<=not rst;

-- Instantiate the Unit Under Test (UUT)

   uut1: Ansteurung PORT MAP (
          clk => clk,
          new_rx_data => new_rx_data,
          rx_data => rx_data,
          ready => ready,
          clkout => clkout,
          reset => reset,
          start => start,
          data_in => data_in,
          command_in => command_in,
          addr_in => addr_in
        );

       uut2: DAC5686 PORT MAP (
          clk => clkout,
          reset => reset,
          start => start,
          data_in => data_in,
          addr_in => addr_in,
          command_in => command_in,
          sdo_5686 => sdo_5686,
          sck_5686 => sck_5686,
          sdi_vdac => sdi_vdac,
          sync_vdac => sync_vdac,
          reset_vdac => reset_vdac,
          ldac_5686 => ldac_5686,
          ready => ready
        );
       uut3: avr_interface PORT MAP (
                     clk=> clk,
                     rst=>n_rst,
                     cclk=>cclk,
                     rx=>rx,
                     rx_data=>rx_data,
                     new_rx_data=>new_rx_data
                     );

       end Behavioral;
```

Anhang C: Qt-Programm

```
#include "mojoserial.h"
#include "ui_mojoserial.h"
#include <QSerialPort>
#include <QSerialPortInfo>
#include <QDebug>



QString rec_message_Buffer="";//Empfangsbuffer
QString rec_message;//vollständige Empfangene Nachricht

QString USB_Port; //Bezeichnung des Ports

QSerialPort serial; //USB-Port


MojoSerial::MojoSerial(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MojoSerial)
{
    ui->setupUi(this);
     ui->lineEdit->setInputMask("HHHHHH");

    //automatisch Ermittlung des Ports
    foreach(const QSerialPortInfo &serialPortInfo, QSerialPortInfo::available-
Ports()){
          if((serialPortInfo.vendorIdentifier()==10717) && (serial-
PortInfo.productIdentifier()==32769)){
               USB_Port = serialPortInfo.portName();
               qDebug() << USB_Port << "\n";
        }
    }

    //Konfiguration der Schnittstelle:
    serial.setPortName(USB_Port);                           //Arduino COM9 //
Mojo COM3 // Linux: "/dev/ttyACM0"
    serial.setBaudRate(QSerialPort::Baud115200);
    serial.setDataBits(QSerialPort::Data8);
    serial.setParity(QSerialPort::NoParity);
    serial.setStopBits(QSerialPort::OneStop);
    serial.setFlowControl(QSerialPort::NoFlowControl);
    serial.open(QIODevice::ReadWrite);//starte Port

    //readyRead-Signal einbinden:
    QObject::connect(&serial,SIGNAL(readyRead()),this,SLOT(read_serial()));
}


//Fenster schließen:
MojoSerial::~MojoSerial()
{
    delete ui;
    //Port schließen
    serial.close();
}


//Button "Senden" wird betätigt:
void MojoSerial::on_pushButton_clicked()
{
    QString trans_message;
    qDebug() << "SEND";
```

```
    //Eingabe aus Textfeld speichern:

    trans_message = ui->lineEdit->text();

 bool ok;
 QByteArray hex_message("\x00\x00\x00");


 hex_message[0]=trans_message.left(2).toInt(&ok,16);
 hex_message[1]=trans_message.mid(2,2).toInt(&ok,16);
 hex_message[2]=trans_message.right(2).toInt(&ok,16);
 serial.write(QByteArray::fromRawData(hex_message,1));
 serial.write(QByteArray::fromRawData(hex_message,2));
 serial.write(QByteArray::fromRawData(hex_message,3));
    //Eingabe konvertieren und senden
    //serial.write(test);
   //serial.write(trans_message.toUtf8());
   // serial.write(trans_message.toUtf8());

    //Textfeld löschen
    ui->lineEdit->setText("");

     //qDebug() << "Hex: "<<test;
    qDebug() << trans_message.toUtf8();
}

//Daten sind lesebereit:
void MojoSerial::read_serial()
{
    qDebug() << "READ:";
    //Buffer füllen:
    rec_message_Buffer += serial.readAll();
    //bei Zeilenumbruch den Buffer speichern und leeren:
    if(rec_message_Buffer.contains("\n")){//
        rec_message = rec_message_Buffer;
        rec_message.remove("\n");
        rec_message_Buffer="";
    }
    qDebug() <<rec_message;

    ui->lineEdit_2->setText(rec_message);
}
```