Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Bachelorarbeit

Desgin und Implementierung eines FPGA-Event-Recorders mithilfe der freien IceStorm-Toolchain

Studienrichtung: Technische Informatik

Domenik Müller

Hochschule für angewandte
Wissenschaften Augsburg

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon +49 821 55 86-3450
Fax +49 821 55 86-3499

Verfasser der Diplomarbeit
Domenik Müller
Am Eser 3
86150 Augsburg
Telefon +49 821 44 92 57 54
domenik.mueller@hs-augsburg.de

Prüfer: Prof. Dr. Hubert Högl
Zweitprüfer: Prof. Dr. Alexander von Bodisco

Abgabedatum: 20.06.2018

Zusammenfassung

In der vorliegenden Arbeit wird der Entwurf und die Implementierung eines FPGA-basierten Event-Recorders mithilfe der Open-Source-Toolchain IceStorm beschrieben. Ähnlich wie bei einem Logikanalysator werden digitale Signaländerungen an den Eingängen erfasst, allerdings werden die Eingangssignale nicht wie bei einem Logikanalysator kontinuierlich übertragen, sondern werden bereits zur Erfassungszeit nach relevanten Eingangskombinationen gefiltert. Ein Filterung nach vordefinierten Events bietet sich vor allem für Timinganalysen bei relativ geringer Signaldichte an und kann die zeitliche Auflösung der untersuchten Signale verbessern. Die Implementierung des Event-Recorders erfolgt auf einem Raspberry Pi Zero mit IceZero FPGA-Shield und wird vollständig mit den von der IceStorm-Toolchain zur Verfügung gestellten Open-Source-Tools und Komponenten umgesetzt.

Abstract

The work in hand describes the design and implementation of a FPGA based event recorder using the open source toolchain IceStorm. Similar to a logic analyzer it records logic level signal changes, but in contrast to a logic analyzer it does not provide a continuous stream of the input signals. Instead the signals are filtered at capture time to match only relevant input combinations. Using predefined events to filter the input stream is especially suitable for timing analysis of signals with low event density and can improve the timing resolution of the analyzed signals. The implementation of the event recorder is done on a Raspberry Pi Zero with a IceZero FPGA shield and is realized completely with the tools and components provided by the open source toolchain IceStorm.

Inhaltsverzeichnis

1. Einführung

Mikrocontroller werden heute in Gebrauchsgegenständen aller Art verbaut und werden den Anforderungen entsprechend immer leistungsstärker und damit unter anderem auch schneller. Selbst einfache Mikrocontroller arbeiten oft mit einer Geschwindigkeit im mehrstelligen Megaherz Bereich (sprich: mehrere Millionen Takte pro Sekunde). Im Unterschied zu klassischen PC-Systemen werden an Mikrocontroller oft Echtzeit-Anforderungen gestellt, das heißt Ergebnisse müssen zuverlässig innerhalb einer vorbestimmten Zeitspanne geliefert werden[1]. Dabei wird auch die Hardware von Mikrocontrollern zunehmend komplexer und es werden vermehrt Mehrprozessor-Systeme verwendet, die angepasste und mitunter unübersichtlichere Programmiertechniken erfordern.
Dementsprechend werden für die Entwicklung von Mikrocontroller-Systemen (aber auch von digitalen Systemen im allgemeinen) Werkzeuge benötigt, mit denen Signale mit hoher zeitlicher Auflösung erfasst und analysiert werden können.
Ergänzend zu Simulations- und Software-gestützten Verfahren wird diese Aufgabe meist von Logikanalysatoren erfüllt, die die an den Eingängen anliegenden Spannungen mit einer festen Frequenz erfassen und die Daten dann zum Beispiel an einen PC übertragen, an dem sie ausgewertet werden können.
In der vorliegenden Arbeit wird eine spezielle Form von Logikanalysator entworfen, bei der ein Teil der Auswertung bereits auf dem Logikanalysator durchgeführt wird. Das Eingangssignal wird auf bestimmte - vom Benutzer definierte - Signaländerungen untersucht und nur relevante Signaländerungen (“Events”) werden an den Benutzer weitergereicht.
Dieses Vorgehen bietet sich vor allem dann an, wenn der Signalverlauf grundsätzlich bekannt ist und der Fokus der Analyse auf den exakten zeitlichen Abbildung des erwarteten Signalverlaugs liegt.
Ein Beispiel wären die Ausgänge eines Mikrocontrollers, bei dem bewusst bestimmte Kombinationen gesetzt werden um den Start und das Ende von Funktionen im Quellcode zu signalisieren. Da das Setzen von GPIO-Pins meist in einem einzigen CPU-Takt ausgeführt werden kann, können so zuverlässige Aussagen zur Laufzeit von Funktionen, oder bei periodischer Ausführung auch zur zeitlichen Fluktuation der Funktionsausführung (“Jitter-Analyse”) getroffen werden.

1.1 Zielsetzung

Für die Implementierung des Event-Recorders wurden folgende technische Ziele angestrebt:

- Es soll der logische Pegel von 8 bis 16 Eingangs-Pins abgefragt werden und die Eingangsdaten sollen mit einem stabilen Zeitstempel versehen werden.
- Die zeitliche Auflösung der Aufnahme soll im Megaherz-Bereich liegen
- Bestimmte Eingangskombinationen sollen in Textform definiert, und bei der Aufnahme als Events erkannt werden
- Zur Steuerung der Aufnahme soll ein Kommandozeilentool zur Verfügung stehen, mit dem auch die aufgenommenen Daten in Textform abgespeichert werden können.

1.2 Motivation

Die Arbeit schließt thematisch an die Bachelorarbeit "Ein universales, rekonfigurierbares und freies USB-Gerät zur Timing-, Protokoll-, Logik- und Eventanalyse von digitalen Signalen" von Andreas Müller und einer darauf folgenden Projektarbeit an.
In der Bachelorarbeit wurde eine Hardware-Platine namens "USB-TPLE" mit USB-Schnittstelle, einem CPLD-Chip von Altera und einem Atmega Mikrocontroller für die selbe Zielsetzung entworfen, und mit der Software-Implementierung begonnen[2].
Im nachfolgenden Semester-Projekt "Logikanalysator mit AVR Mega32U4 und Altera MAX CPLD" im Wintersemester 2013/14 wurde die Software-Implementierung ausgebaut und eine funktionsfähige Konfiguration für den CPLD-Chip entwickelt.

Im folgenden wird allerdings ein anderer Ansatz für die Umsetzung verfolgt:

Verwendung von käuflich verfügbarer Hardware
: Anstatt der selbst entworfenen Platine soll aus Gründen der Vefügbarkeit und um die Einstiegshürde für Benutzer zu verringern ein käuflich erwerbbares Produkt verwendet werden.
Die Verwendung käuflicher Hardware soll außerdem die Gesamt-Komplexität des Projekts reduzieren einen stärkeren Fokus auf Grundfunktionalität ermöglichen.

Verwendung eines FPGAs
: Um größere Flexibilität bei der Implementierung zu ermöglichen wird ein Field Programmable Gate Array (FPGA) anstatt des Complex Programmable Logic Device (CPLD) verwendet (eine detailliertere Erklärung findet sich im Kapitel Design).

Neben Verfügbarkeit und Flexibilität des Designs soll vor allem ein weiterer Grundsatz bei der Implementierung verfolgt werden:

Verwendung von Open-Source Software und Hardware
: Bereits die Arbeit von Andreas Müller wurde unter einer Open-Source-Lizenz veröffentlicht und es wurden alle Projekt-Quellen und Ressourcen (einschließlich des Hardwaredesigns) öffentlich verfügbar gemacht.
Dieser Ansatz soll hier weiter verfolgt werden, dementsprechend werden alle im Rahmen dieser Arbeit entstandenen Dokumente unter der LGPL3-Lizenz veröffentlicht (siehe Anhang B).
Ausserdem steht mit dem Projekt "IceStorm" erstmals auch eine Open-Source Software-Toolchain zur Programmierung von FPGA-Chips zur Verfügung, wodurch eine vollständige Open-Source Implementierung möglich wird. (In der vorliegenden Arbeit mit Ausnahme der proprietären Komponenten des Raspberry Pi Zero).

In Kombination mit der preisgünstigen Hardware bietet die IceStorm-Toolchain eine interessante Alternative zu den Angeboten der großen FPGA-Hersteller wie Xilinx oder Intel (ehemalig Altera), insbesondere für Lehrzwecke und kleinere Projekte.

1.3 Abgrenzung von bestehenden Lösungen zur Logikanalyse

Es ist eine Vielzahl von kommerziellen Logikanalysatoren am Markt verfügbar. Allerdings bieten selbst sehr flexible Geräte wie zum Beispiel die Discovery Serie von Digilent nicht die gewünschte Funktionalität der Event-Filterung zur Erfassungszeit mit der Möglichkeit die so gewonnenen Daten in einen Text- bzw. Kommandozeilen-basierten Workflow einzubetten [1].

Von kommerziellen Produkten abgesehen gibt es auch einige Open-Source Logikanalysatoren. Für diese Arbeit relevant sind hier vor allem:

SUMP2 ist eine Verilog-basierte Logikanalysator-Implementierung mit einer zugehörigen - in Python implementierten - grafischen Benutzeroberfläche. Es existieren angepasste Varianten von SUMP2 die ohne weitere Modifikationen auf dem auch in dieser Arbeit verwendeten iCE40-FPGA-Chip lauffähig sind[4].

Open Bench Logic Sniffer ist ein Open-Source Hardware-Produkt das auf einem Xilinx Spartan 3E FPGA basiert und eine weiterentwickelte Variante von SUMP2 verwendet. Die "Demon core" betitelte Weiterentwicklung ist insbesondere deshalb interessant, da mit ihr detailliertere Triggerbedingungen definiert werden können, und so zum Beispiel zeitliche und logische Abläufe von Eingangssignalen als Trigger abgebildet werden können. Hierauf soll im Kaptiel Aussicht noch einmal eingegangen werden.

Das verwendete SUMP2 Datenübertragungsformat wird zum Teil auch von anderen Anwendungen unterstützt, so kann zum Beispiel der Java-Client Jawi[5] oder Pulseview[6] (ein Qt-Frontend der libsigrok-Biliothek) als grafische Benutzeroberfläche verwendet werden.
Beide Varianten verwenden zur Datenübertragung eine serielle Schnittstelle (UART), die - zumindest bei Verwendung von geläufigen Baud-Raten - die Übertragungsgeschwindigkeit stark einschränkt. Ebenso sind beide Varianten konzeptionell für die Aufnahme festgelegter und relativ kurzer Sampling-Zeiten ausgelegt und unterstützen — wie die kommerziellen Produkte — keine Event-Filterung zur Erfassungszeit.

Eine Anpassung des SUMP2 Projektes wurde in Erwägung gezogen, aber aufgrund der zum Teil recht hohen Code-Komplexität und der strukturellen Unterschiede nicht durchgeführt.

1.4 Aufbau der Arbeit

Im folgenen Kapitel Design werden zunächst die nötigen technischen Grundlagen für die Umsetzung des Projekts besprochen, anschließend wird auf getroffene Desginentscheidungen bei der Auswahl der Hardware und Software eingegangen, und ein kurzes Implementierungs-Beispiel mit der IceStorm-Toolchain erläutert. Das Kaptiel Implementierung beschreibt di nötigen Anpassungen bestehender Software und die Entwicklung neuer Softwarekomponenenten bei der Durchführung des Projektes. Im Kapitel Anwendungsfall: Jitter-Analyse von Software-generierten MIDI-Clock Signalen wird die Benutzung des Event-Recorder anhand eines konkreten Beispiels besprochen. Es folgt ein Fazit in dem der Status des Projekts und die Umsetzung rekapituliert werden und abschließend wird im Kapitel Aussicht auf Optimierungsmöglichkeiten und weiteres Vorgehen eingegangen.

[1] Geräte der Discovery-Serie können durch eine API z.B. in Python geskriptet werden, eine kontinuierliche "Event-Erkennung" scheint aber nicht ohne weiteres möglich (siehe z.B. folgender Foreneintrag[3])

2. Design

2.1 Theoretische Grundlagen

Zunächst sollen die grundlegenden Eigenschaften der erwarteten Eingangssignale definiert und näher beschrieben werden.

2.1.1 Zeit- und wertdiskrete digitale Signale

Bei den Eingangssignalen des Event-Recorders handelt es sich um die Ausgänge von Mikrocontrollern oder anderer digitaler Schaltungen und damit um digitale Signal. Digitale Signale sind durch zwei grundlegende Eigenschaften charakterisiert, sie sind:

- **zeitdiskret**, und
- **wertdiskret**

Wertdiskret bedeutet, dass das Signal nur genau einen Wert aus einer festgelegten Anzahl möglicher Wertezustände annehmen kann. In den meisten Fällen beschränkt sich der Wertebereich auf die Binärwerte “1” oder “0”, das heißt ein digitales Signal ist zum Beispiel bei 0,1 Volt Spannung “0” und bei 3,2 Volt “1”, wohingegen ein analoges Signal alle möglichen Werte zwischen 0 und 3,3 Volt annehmen kann.

Zeitdiskret bedeutet, dass ein digitales Signal “nur zu bestimmten periodischen Zeitpunkten definiert ist bzw. nur dann eine Veränderung im Signalwert aufweist”[7], das heißt dass das Signal bei der Erfassung mit einem festen “Zeitraster” abgetastet wird und zwischen den Rasterpunkten einen festen Wert behält.

Bei der Erfassung digitaler Signale ist es nötig, dass die Abtastfrequenz nach Nyquist-Shannon-Abtasttheorem mindestens doppelt so hoch als die maximale Frequenz des untersuchten Signals sein muss, damit keine Informationen aus dem Ausgangsignal verloren gehen (vgl. [7]).

Es ist zu beachten, dass es sich bei der Definition von digitalen Signalen um eine idealisierte Ansicht handelt und dass sich bei realen digitalen Signale oft – vor allem bei hohen Frequenzen – Störeffekte zeigen. Ein Beispiel für einen Störeffekt wäre das “Prellen” eines mechanischen Schalters, bei dem es durch den mechanischen Kontakt zu einem mehrfachen Signalwechsel kommen kann, bevor ein stabiles Signal anliegt. Wenn der Schalter-Zustand

dabei mit vergleichsweise langsamer Geschwindigkeit ausgelesen wird, gehen die deutlich schnelleren Signalwechsel gemäß dem Nyquist-Shannon-Abstasttheorem nicht in das Ausgangssignal ein, bei entsprechend hoher Abtastrate werden sie allerdins mit in Ausgangssignal übernommen und können dann unerwünschte Effekte auslösen.
Bei der in dieser Arbeit verwendeten Hardware sind keine speziellen Vorrichtungen (wie zum Beispiel "Schmitt-Trigger") vorhanden um solchen Effekten entgegen zu wirken, das heißt es wird am Eingang ein für die Analyse aussreichend stabiles Signal erwartet.

Damit die Ergebnisse des Event-Recorders richtig auswertet werden können, muss zu jeder Signaländerung der "diskrete" Zeitpunkt bekannt sein, das heißt das Zeitraster der Abtastung muss numeriert werden, so dass jeder Signaländerung ein eindeutiger Zeitstempel zugeordnet werden kann. Diese "Numerierung" wird umgesetzt, in dem der von einem Oszillator[1] generierte Schaltungstakt mit einer Zählerschaltung aufaddiert wird. Der Zeitstempel entspricht dann einfach dem aktuellen Zählerstand.

EVENT_ID [8 Bit]	INPUT_DATA [8 Bit]	TIMESTAMP [16 Bit]
0x02	0x01	0x0EF8

Tabelle 2.1: Beispielhafte Darstellung eines aufgenommenen Events mit 16-Bit Zeitstempel

Die Bit-Breite des Zählers legt zusammen mit der Geschwindigkeit des Takts die maximal mögliche Aufnahmelänge fest. Der verwendete Zähler hat eine Breite von 46 Bit, womit sich bei einer Taktfrequenz von 100 Mhz zum Beispiel eine Laufzeit von ca. 195 Stunden ergibt[2] (was für die meisten Anwendungsfälle mehr als aussreichend ist).

2.1.2 Definition "Event"

Für diese Arbeit soll unter dem Begriff "Event" ein vom Benutzer festgelegter Signalzustand oder eine Signaländerung an den Eingangspins des Event-Recorders verstanden werden. In den meisten Fällen macht es mehr Sinn eine Signaländerung zu definieren, als einen Zustand, da bei einem anhaltenden Zustand auch das Event kontinuierlich "ausgelöst" wird, und ein dementsprechend hohes Datenvolumen erzeugt.
Es ergeben sich folgende Definitionsmöglichkeiten für die einzelnen Eingangs-Pins:

- "0": niedriger logischer Pegel
- "1": hoher logischer Pegel
- "u": steigender Pegel
- "d": fallender Pegel
- "x": beliebiger Zustand ("don't care")

Zusätzlich wäre eine Kombination von "u" und "d" denkbar, die auf beliebige Signaländerungen reagiert. Eine Verkettung dieser Möglichkeiten bei der jedem Eingangspins ein Zustand zugewiesen wird soll als "Event-Trigger" – also als Auslöser eines bestimmten Events – bezeichnet werden.
Dies entspricht im Wesentlichen der Definition von *Trigger* die bei "traditionellen" Logikanalysatoren verwendet wird, allerdings mit dem Unterschied dass der Trigger bei einem

[1] Eigener Hardware-Baustein, der eine konstant auf- und abschwingende Spannung erzeugt und damit zur Taktung digitaler Schaltung verwendet werden kann

[2] Berechnung: $2^{46} * \frac{1}{100Mhz} = 2^{47} * 10ns$

"traditionellen" Logikanalysator die eigentliche Aufnahme einmalig auslöst, und dann bis zum Ende der Aufnahme nicht mehr von Bedeutung ist, während ein Event-Trigger als Teil der Aufnahme kontinuierlich überprüft werden muss und erst bei Erfüllung der Triggerbedingung überhaupt Ausgangsdaten generiert werden.
Es bietet sich an einem Event zusätzlich bestimme Funktionen zuweisen zu können, wie zum Beispiel das Starten[3] und Stoppen der Event-Erkennung, oder das Wechseln in einen zusätzlichen Modus, bei dem auf alle Signaländerungen reagiert wird.
Wie in der Einführung erwähnt soll die Definition der Events in Text-Form möglich sein und kann dann z.B. folgendermaßen aussehen:

```
# event configuration
events:
  - start:
          trigger: uxxxxxxx
          func: start

  - stop:
          trigger: dxxxxxxx
          func: stop

  - event1:
          trigger: 1uxxxxxx

  - event2:
          trigger: 1xuxxxxx
          func: dump_begin
...
```

[3] Die Erkennung eines Start-Events erfordet folglich, dass auch im "Ruhezustand" eine Event-Erkennung durchgeführt wird

2.2 Überblick der benötigten Hard- und Software-Komponenten

2.2.1 Datenerfassung: FPGA

Wie im vorherigen Kapitel beschrieben wird für die Datenerfassung eine Zählerschaltung benötigt, die einen stabilen Zeitstempel liefern kann. Vorraussetzung dafür ist, dass der Zähler kontinuierlich läuft und nicht durch andere Vorgänge unterbrochen werden kann. Dies ist bei PC-Systemen oder auch Mikrocontrollern nicht ohne weiteres möglich, da die Programmausführung zu jedem Zeitpunkt von Betriebssystem-Funktionen oder Interrupt-Routinen[4] pausiert werden kann.
Deswegen bietet sich hier die Verwendung einer programmierbaren logischen Schaltung wie zum Beispiel eines CPLDs (Complex Programmable Logic Device) oder FPGAs (Field Programmable Gate Array) an, bei dem eine von anderen Komponenten vollkommen unabhängige parallele Ausführung des Zählers möglich ist.
Sowohl CPLD- als auch FPGA-Chips bestehen aus einer Vielzahl von einheitlichen Blöcken ("PLBs") die einfache logische Funktionen abbilden können. Im Vergleich zu logischen Gattern sind die die Blöcke in ihrer Funktion aber frei konfigurierbar. Durch die Vernetzung der so konfigurierten Blöcke können umfangreiche digitale Schaltungen realisiert werden.
FPGAs sind etwas komplexer aufgebaut als CPLDs, dabei aber auch flexibler bei der Vernetzung und enthalten oft zusätzliche Funktionsblöcke wie den in der nachfolgenden Abbildung erkennbaren Block-RAM als Zwischenspeicher für größere Datenmengen oder die PLL-Einheit zur Erzeugung von Taktsignalen mit konfigurierbarer Geschwindigkeit (vgl. [8]).

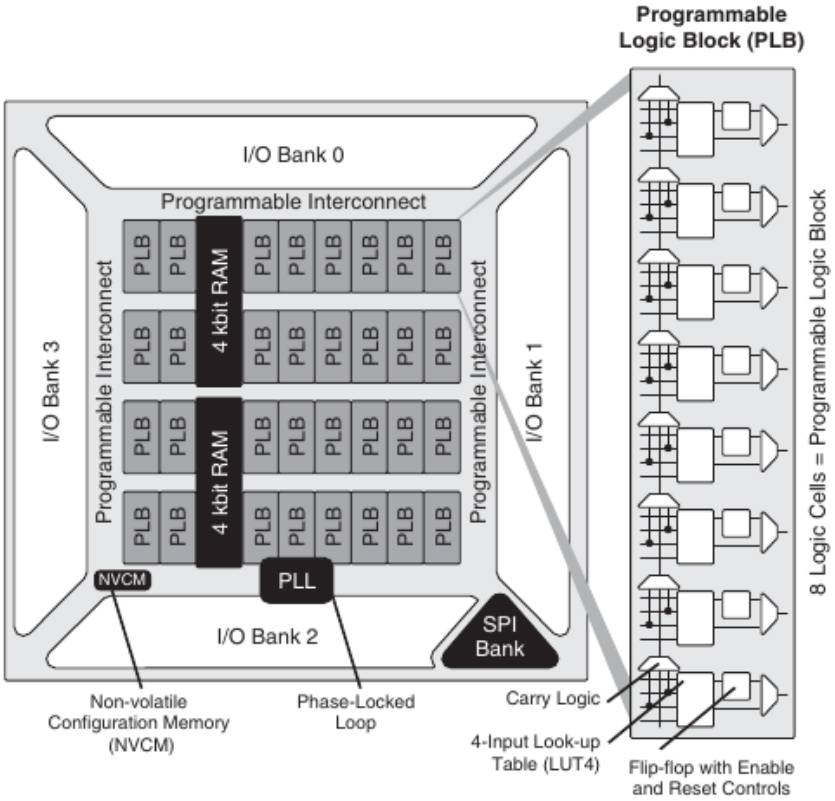

Abbildung 2.1: Aufbau des verwendeten iCE40 FPGAs (Quelle: iCE40 Datasheet[9])

Zusätzlich ist die Anzahl von Funktionsblöcken bei FPGAs üblicherweise deutlich höher als als bei CPLDs, weswegen in dieser Arbeit ein FPGA-Chip verwendet werden soll.

[4]Bei einer Interrupt-Routine wird die aktuelle Programmausführung auf CPU-Ebene pausiert, zum Beispiel um auf Ereignisse externer Hardwaregeräte reagieren zu können

fehlt: GPIO pins

2.2.2 Datenzwischenspeicher: SRAM

Nach der Erfassung müssen die Daten zwischengespeichert werden. Dabei sind vor allem zwei Faktoren entscheidend:

- Die **Geschwindigkeit** des Speichers, die meist den maximalen Datendurchsatz der Gesamtschaltung bestimmt
- Die **Größe** des Speichers, die festlegt wie lange bei hohem Datendatendurchsatz aufgenommen werden kann

Die meisten nicht-flüchtigen Speicher sind aufgrund der unzureichenden Geschwindigkeit nicht für diesen Anwendungszweck geeignet, weswegen sich die Verwendung von RAM-Speicher anbietet.
Bei Mikrocontrollern und kleineren FPGA-Boards werden aus Kostengründen und wegen der unkomplizierten Ansteuerung oft SRAM-Speichereinheiten verbaut. SRAM-Speicher hat meist sehr kurze Zugriffszeiten, allerdings bei einer geringen Speichergröße von wenigen MBit. Der in dieser Arbeit verwendete SRAM-Speicher hat beispielsweise eine Größe von 4 Mbit (512 KB) und könnte damit 62500 Events zwischenspeichern[5].
Bei kontinuierlicher Erfassungs mit einer Abtastrate von 100 Mhz entspricht dies einer Aufnahmezeit von unter einer Millisekunde[6]. Da allerdings keine Eingangssignale erwartet werden, bei denen Events mit einer Frequenz von 100 Mhz auftreten und außerdem bereits bei laufender Aufnahme Events aus dem RAM-Speicher entnommen und weiter übertragen werden, kann der SRAM-Speicher trotzdem im Sinne der Aufgabenstellung als Zwischenspeicher verwendet werden.
Als Alternative könnte DRAM-Speicher verwendet werden, der ein Vielfaches der Speichergröße bietet und damit auch längere Aufnahmen bei hoher Signaldichte ermöglichen würde. DRAM-Speicher erfordert allerdings im Vergleich zu SRAM ein kontinuierliches "Auffrischen" der Speicher-Inhalte durch eine Controller-Einheit und bedeutet deshalb deutlich mehr Aufwand bei der Implementierung.

2.2.3 Datenübertragung: SPI

Nach der Datenerfassung und Zwischenspeicherung werden die Daten an ein externes System übertragen, an dem sie weiterverarbeitet oder ausgewertet werden können. Zur Datenübertragung gibt es eine Vielzahl von Schnittstellen und Protokollen. Dabei werden die Daten im Normalfall "serialisiert", das heißt wenn ein Event aus 64 Bit besteht, werden die Bits nacheinander über eine einzige Datenleitung übertragen. Geläufige serielle Datenübertragungsverfahren sind vor allem UART, I^2C und SPI.

Bei einem **UART (Universal Asynchronous Receiver Transmitter)** werden die Daten über eine Empfangs- und eine Sendeleitung ausgetauscht. Es wird kein eigenes Taktsignal übertragen, weswegen auf beiden Seiten eine feste Übertragungsgeschwindigkeit ("Baud-Rate") eingestellt werden muss. UART-Schnittstellen sind weit verbreitet, bei üblichen Baud-Raten ist die Übetragungsgeschwindigkeit allerdings relaitv gering (vgl. [10]).

[5] Bei einer angenommenen Event-Größe von 64-Bit, und unter der Annahme dass der SRAM-Speicher ausschließlich zum Zwischenspeichern von Events verdwendet wird

[6] $62500 * 10ns = 0.625ms$

I^2C (Inter-Integrated Circuit) ist ein synchroner Datenbus, bei dem eine eigene Leitung für den Takt und eine Datenleitung verwendet wird. I^2C arbeitet nach dem Master-Slave-Prinzip, das heißt es können auch mehrere Geräte miteinander kommunizieren. I^2C untertützt Übertragungsraten bis zu 5 Mbit/s (unidirektional, vgl. [11]).

SPI (Serial Peripheral Interface) ist wie I^2C ein synchroner Datenbus nach dem Master-Slave-Prinzip. Neben der Leitung für den Takt wird eine Daten-Leitung in Senderichtung und eine Datenleitung in Empfangsrichtung verwendet. Zusätzlich wird pro Gerät eine "Chip-Select" Leitung benötigt, um die Geräte adressieren zu können. SPI kann Übertragungsraten bis in den mehrstelligen Megabit-Bereich ermöglichen (vgl. [12]).

Aufgrund der hohen Übertragungsrate und der relativ unkomplizierten Implementierungsmöglichkeiten soll SPI für die Datenübetragung verwendet werden.

2.2.4 Steuerung der Aufnahme und sequentielle Programmabläufe

Neben der reinen Datenerfassung und -Übertragung werden noch Komponenten zur Steuerung und zur Kontrolle des Aufnahmevorgangs benötigt.
Grundsätzlich können die benötigten Vorgänge- und Zustände (zum Beispiel das Starten und Stoppen der Aufnahme) direkt im FPGA umgesetzt werden. Als Kommunikationsweg zur Steuerung und Konfiguration kann dann wiederum SPI verwendet werden.
Bei längeren oder komplexeren Programmabläufen die keine zeitkritische Ausführung erfordern bietet sich die Verwendung eines zusätzlichen Mikrocontrollers an. Da die meisten PC-Systeme keine programmierbaren GPIO-Pins zur Verfügung stellen, kann ein Mikrocontroller auch als Brücke zwischen FPGA und Anwendersystem fungieren, und zum Beispiel die vom FPGA erfassten Daten über einen USB-Port zur Verfügung stellen.
Davon abgesehen wird auch auch die entsprechende Hardware- und Software-Infrstruktur benötig um das FPGA und den Mikrokonroller zu programmieren und zu konfigieren.

2.3 Auswahl der Software-Toolchain: IceStorm

2.4 Auswahl der Hardware

2.4.1 IceZero FPGA-Shield (iCE40HX4K)

FPGA: iCE40 vielzahl boards: z.B. https://embeddedmicro.com/products/mojo-v3 -> SDRAM

-> keine Opensource-Toolchain!

iCE40: erst: https://www.olimex.com/Products/FPGA/iCE40/iCE40HX1K-EVB/open-source-hardware

dann icezero:

- formfaktor

2.4.2 Raspberry Pi Zero W

Controller: pi: preis, vielfältigkeit, formfaktor

2.5 Beispiel: Von der Synthese bis zum Bitstream mit der IceStorm-Toolchain

3. Implementierung

...

3.1 Portierung des Tools zum Flashen des Bitstreams (icoprog)

...

3.2 Portierung und nötige Anpassungen des Verilog-SoCs (icosoc)

Bla fasel...

3.3 Implementierung des Event-Recorder Moduls

3.3.1 Bus-Schnittstelle

3.3.2 Triggerlogik

3.4 Implementierung eines SPI-Slave-Moduls

Bla fasel...

3.5 Zusammenführung der Module als Icosoc-Projekt

3.6 Implementierung des textbasierten Benutzerinterfaces

4. Anwendungsfall: Jitter-Analyse von Software-generierten MIDI-Clock Signalen

Als Beispiel für ein zeitkritisches Signal sollen im Folgenden mehrere MIDI-Clock Signale mit dem Event-Recorder untersucht werden, und dabei der allgemeine Ablauf einer Event-Aufnahme und der nachfolgenden Analyse geschildert werden. Das MIDI-Clock Signal wird in den untersuchten Fällen software-basiert – auf einem normalen Anwender-System – generiert, deswegen ist von einer messbaren zeitlichen Fluktuation ("Jitter") auszugehen, das heißt die Clock-Signale treten nicht exakt zum erwarteten Zeitpunkt sondern leicht zeitlich verfrüht oder verspätetet auf.
MIDI ist ein 1983 spezifizierter Standard für den Austausch von Steuerinformation zwischen elektronischen Musikintrumenten und wird trotz einiger signifikanter Limitierungen seit über 35 Jahren nahezu unverändert für praktisch alle elektronischen Musikinstrumente im professionellen und Hobby-Bereich verwendet. MIDI bietet mit der "MIDI-Clock" eine Möglichkeit mehrere Geräte zeitlich zu synchronisieren. So könnte zum Beispiel ein Synthesizer mit der Musiksoftware auf einem PC synchronisiert werden, um tempo-abhängige Arpeggios[1] zu spielen.
Dabei verwendet MIDI zur Datenübertragung das gleiche Protokoll wie ein UART (ohne Parity-Bit) bei einer Übertragungsgeschwindigkeit 31250 Bit/s. Die meisten MIDI-Nachrichten setzen sich aus einem Stautsbyte und zwei darauf folgenden Datenbytes zusammten, für die MIDI-Clock werden allerdings nur einzelne Bytes benötigt.
Zuerst wird ein Start-Byte (0xFA) übertragen, dann wird – abhängig vom Tempo – 24 mal pro Viertelnote ein "Clock-Tick" (0xF8) gesendet, und abschließend ein Stopp-Bit (0xFC).
Eine Implementierung in Python könnte folgendermaßen aussehen:

```
# MIDI CLOCK TEMPO (beats per minute)
BPM = 80

# define clock messages
clock_start = [0xFA]
clock_tick  = [0xF8]
clock_stop  = [0xFC]
```

[1] Ein Akkord bei dem die einzelnen Töne nicht gleichzeitig, sondern versetzt nacheinander gespielt oder ausgelöst werden

```
[...]
    # calculate clock period
    clock_period = 60 / (BPM * 24)

    # send start byte
    midiout.send_message(clock_start)

    # run
    while (True):
        midiout.send_message(clock_tick)
        time.sleep(clock_period)
```

4.1 Test-Setup: USB-Midi mit Teensy LC

Um das MIDI-Signal auswerten zu können wird ein Mikrokontroller ("Teensy LC") verwendet, der per USB an einen PC angeschlossen werden kann, und über die USB-Schnittstelle ein MIDI-Gerät simuliert. Das heißt am PC wird eine MIDI-Schnittstelle erzeugt, und die MIDI-Daten werden direkt zum Mikrocontroller übertragen. Der Mikrocontroller wartet auf das Start-Byte und setzt einen GPIO-Pin auf "1" um dem Event-Recorder den Anfang der Aufnahme zu signalisieren. Danach wird bei jedem empfangenen Clock-Tick der ein zweiter GPIO-Pin kurz auf "1" gesetzt und anschließend wieder auf "0". Dies soll beim Event-Recorder als Event erkannt werden. Beim Empfangen des Stopp-Bytes word der erste GPIO-Pin auf "0" gesetzt und die Aufnahme soll beendet werden.
Der Mikrocontroller kann mit der Arduino-IDE programmiert werden und es steht eine MIDI-Bibliothek zur Verfügung, was eine kurze und unkomplizierte Umsetzung erlaubt:

```
[...]
// midi clock start handler (0xFA)
void onStart() {
  digitalWrite(0, HIGH);
}

// midi clock tick handler (0xF8)
void onClock() {
    digitalWrite(1, HIGH);
    digitalWrite(1, LOW);
}

// midi clock stop handler (0xFC)
void onStop() {
  digitalWrite(0, LOW);
}

void setup() {
  // pin setup
  [...]
  // register midi handlers
  usbMIDI.setHandleStart(onStart);
  usbMIDI.setHandleStop(onStop);
  usbMIDI.setHandleClock(onClock);
}
```

```
void loop() {
  usbMIDI.read();
}
```

4.2 Einrichten des Projekts

Die vollständige Einrichtung des Raspberry Pi Zero W soll hier aus Platzgründen nicht beschrieben werden, es steht aber eine detailliertere Anleitung im Projekt-Wiki zur Verfügung. Der grundsätzliche Ablauf ist wiefolgt:

1. Download, Kompilieren und Installation der IceStorm-Toolchain auf den Anwender-PC (Linux-System)

   ```
   # siehe http://www.clifford.at/icestorm/#install
   ```

2. Download, Kompilieren und Installation der RISC-V Toolchain

   ```
   git clone git@github.com:cliffordwolf/picorv32.git
   cd picorv32
   make download-tools
   make -j\$(nproc) build-tools
   ```

   Das Kompilieren der RISC-V gcc-Toolchain kann je nach Rechenleistung mehrere Stunden dauern.

3. Download des Git-Projekts auf den Anwender-PC (Linux-System)

   ```
   cd ..
   git clone https://github.com/dm7h/icozsoc.git
   ```

4. Einrichten der SSH-Verbindung zum Raspberry Pi Zero W

   ```
   # siehe zum Beispiel: https://www.raspberrypi.org/documentation/remote-acc
   # Exportieren des SSH-Hostsi, z.B.:
   export SSH_RASPI=pi@zero
   # eine SSH-Verbindung sollte dann ohne Passwort-Abfrage m\"oglich sein:
   ssh \textdollar SSH_RASPI
   ```

5. Download, Kompilieren und Installation des icozctl Git-Projekts Zur Installation des icozctl-Tools wird auf dem Raspberry Pi folgendes ausgeführt:

   ```
   git clone https://github.com/dm7h/icozctl
   cd icozctl
   sudo make install
   ```

6. Generieren und Programmieren des Bitfiles mithilfe des zur Verfügung gestellten Makefiles Wenn icozctl erfolgreich installiert wurde, kann auf dem Anwender-PC das FPGA-Bitstream und die IcoSoc-Anwendung kompiliert und via Raspberry Pi auf die Hardware geflasht werden:

   ```
   cd ../icozsoc/examples/event-recorder/
   make prog_flash
   ```

4.3 Konfiguration der Event-Trigger

Um die Events zu konfigurieren wird zunächst auf das Raspberry Pi gewechselt. Im Ordner des icozctl-Tools befindet sich die Event-Konfigurations-Datei “config.yml”. Für die Aufnahme wird nicht zwangsläufig eine Event-Erkennung benötig, es kann aber aber trotzdem ein Start-, Stopp- und Clock-Event definiert werden um unnötige Aufnahmedaten zu minimieren:

```
# event configuration
events:
  - start:
          trigger: u
          funcion: start

  - stop:
          trigger: d
          function: stop

  - clk_up:
          trigger: 1u

  - clk_down:
          trigger: 1d
```

4.4 Durchführen der Event-Aufnahme

Die Auswertung soll für Anschauungszwecke das VCD-Dateiformat verwenden. Die Aufnahme kann dann mit folgendem Befehl gestartet werden:

```
icozctl -c config.yml -o midi_clock.vcd
```

Das Programm wartet nun auf Eingangsdaten, die zum Beispiel durch das Starten der in Python implementieren Midi-Clock oder durch ein Musikprogramm geliefert werden können. Die Aufnahme kann zu jedem Zeitpunkt durch die Tastenkombination Strg+c beendet werden. Das Ergebnis der Aufnahme kann zum Beispiel mit gtkwave grafisch dargestellt und überprüft werden: [screenshot]

4.5 Analyse der Ergebnisse

Für die Analyse der Daten kann zum Beispiel ein Python-Skript verwendet werden. Um Aussagen über den Jitter des Signals machen zu können bietet sich die Generierung eines Histogramms an, bei dem auf der x-Achse die Abweichung zum erwarteten Clock-Signal und auf der y-Achse die Häufigkeit der Abweichung dargestellt wird.

Für das Einlesen der VCD-Datei wird das Pythong-Modul Verilog_VCD verwendet:

```
import Verilog_VCD
vcd = Verilog_VCD.parse_vcd('midi_clock.vcd')
```

Im VCD-Dateiformat wird jedem Signal ein Symbol als Abkürzung zugewiesen, auf die Daten des ersten Signals der Datei kann zum Beispiel mit der Abkürzung "!" zugegriffen werden. Die Zeitstempel sind in der Python-Liste unter dem Index "tv"("time value") abrufbar. Der zeitliche Abstand zum jeweils vorhergehenden Event (Zeit-Delta) kann folgendermaßen berechnet werden:

```
# get the first time value
last = vcd["!"]["tv"][0][0]

# calculate time deltas
deltas = []
for item in vcd["!"]["tv"][1:]:
    deltas.append(item[0]-last)
    last = item[0]
```

Die Daten werden in ein numpy-Array umgwandelt und es wird die Differenz zum erwarteten Zeit-Delta gebildet. Anschließend werden sie in eine pandas-Zeitserie konvertiert, wodurch automatisch statistisch relevante Kennwerte wie der Mittelwert und die Standardabweichung berechnet und angezeigt werden können:

```
np_deltas = np.array(deltas)
expected_delta = 31250000 # expected clock period in nanoseconds
np_deltas -= expected_delta
print(deltas_series.describe())
```

Abschließend wird mithilfe der matplotlib-Bibliothek ein Histogramm in Millisekunden-Auflösung erzeugt und angezeigt:

```
(deltas_series/pd.Timedelta(milliseconds = 1)).hist()
[...]
plt.show()
```

4.5.1 Software MIDI-Clock: Python-Implementierung

4.5.2 Software MIDI-Clock: Renoise

4.5.3 Hardware MIDI-Clock: Midipal

[buildern]

5. Fazit

Bla fasel...

6. Aussicht

Bla fasel...

Abkürzungen

I²C Inter-Integrated Circuit. 8, 9, 27, *Glossary:* I²C

API Application Programming Interface. 3, 27, *Glossary:* API

CPLD Complex Programmable Logic Device. 2, 7, 27, *Glossary:* CPLD

FPGA Field Programmable Gate Array. 2, 7, 27, *Glossary:* FPGA

GPIO General Purpose Input / Output. 9, 27, *Glossary:* GPIO

MIDI Musical Instrument Digital Interface. 19

PLL Phase-locked loop. 7, 27, *Glossary:* PLL

RAM Random-Access Memory. 8

SPI Serial Peripheral Interface. 8, 9, 27, *Glossary:* SPI

UART Universal Asynchronous Receiver Transmitter. 3, 8, 19, 27, *Glossary:* UART

VHDL Very High Speed Integrated Circuit Hardware Description Language. 27, *Glossary:* VHDL

Glossar

I²C Synchroner, serieller Datenbus nach dem Master-Slave-Prinzip, der Übertragunsraten bis zu 5 Mbit/s ermöglicht (vgl. [11]). 9, 27

API Schnittstelle zur Anwendungsprogrammierung. Erlaubt zum Beispiel die Verwendung von Programmkomponeneten durch eine exteren Skript-Sprache wie Python (vgl. [13]). 27

CPLD Im Vergleich zu FPGAs einfacher aufgebaute programmierbare logische Schaltungen (vgl. [14]). 2, 7, 27

FPGA Ein rekonfigurierbarer Chip dessen Schaltungsstruktur in einer Hardwarebeschreibungssprache (wie VHDL oder Verilog) frei programmierbar ist (vgl. [15]). 2, 7, 27

GPIO Frei programmierbarer Kontakt der zum Beispiel für das Ansprechen externer Hardwarekomponenten verwendet werden kann (vgl. [16]). 27

PLL Elektronische Schaltungsanordnung die die Frequenz eines veränderbaren Oszillators beeinflussen kann. Wird bei FPGAs meist zur Erzeugung von Taktsignalen mit einstellbarer Geschwindkeit verwendet (vgl. [17]). 27

SPI Synchroner, serieller Datenbus für Datenübertragungen nach dem Master-Slave-Prinzip mit dem vergleichsweise hohe Datendurchsätze möglich sind (vgl. [12]). 9, 27

UART Schnittstelle zur asynchronen, seriellen Datenübertragung (vgl. [10]). 8, 27

Verilog Wortkreuzung aus "Verification" und "Logic". Hardwarebeschreibungssparche für Programmierung und Simulation von FPGAs und CPLDs die in den USA geläufiger ist als VHDL (vgl. [18], siehe auch [19] zur Namesnherkunft). 3

VHDL Vor allem in Europa verbeitetete Hardwarebeschreibungssprache für die Simulation und Programmierung von FPGAs und CPLDs (vgl. [20]). 27

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] *Echtzeit – Wikipedia*, [Online; accessed 1. Jun. 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Echtzeit.

[2] A. Müller, „Ein universales, rekonfigurierbares und freies USB-Gerät zur Timing-, Protokoll-, Logik- und Eventanalyse von digitalen Signalen“, Bachelorarbeit, Hochschule Augsburg, 2010.

[3] *Advanced triggering and buffer filling*, [Online; accessed 31. May 2018], Mai 2018. Adresse: https://forum.digilentinc.com/topic/9488-advanced-triggering-and-buffer-filling.

[4] *SUMP2 – 96 MSPS Logic Analyzer for $22*, [Online; accessed 31. May 2018], Okt. 2016. Adresse: https://blackmesalabs.wordpress.com/2016/10/24/sump2-96-msps-logic-analyzer-for-22.

[5] *Open Bench Logic Sniffer - DP*, [Online; accessed 31. May 2018], Juni 2016. Adresse: http://dangerousprototypes.com/docs/Open_Bench_Logic_Sniffer.

[6] *Openbench Logic Sniffer - sigrok*, [Online; accessed 31. May 2018], Mai 2018. Adresse: https://sigrok.org/wiki/Openbench_Logic_Sniffer.

[7] *Digitalsignal – Wikipedia*, [Online; accessed 4. Jun. 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Digitalsignal.

[8] *Programmierbare logische Schaltung – Wikipedia*, [Online; accessed 6. Jun. 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Programmierbare_logische_Schaltung.

[9] *iCE40 LP/HX Family Data Sheet*, Lattice Semiconductor Corp., 2017.

[10] *Universal Asynchronous Receiver Transmitter – Wikipedia*, [Online; accessed 31. May 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Universal_Asynchronous_Receiver_Transmitter.

[11] *I²C – Wikipedia*, [Online; accessed 7. Jun. 2018], Juni 2018. Adresse: https://de.wikipedia.org/wiki/I%C2%B2C.

[12] *Serial Peripheral Interface – Wikipedia*, [Online; accessed 31. May 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Serial_Peripheral_Interface.

[13] *Programmierschnittstelle – Wikipedia*, [Online; accessed 31. May 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Programmierschnittstelle.

[14] *Complex Programmable Logic Device – Wikipedia*, [Online; accessed 26. May 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Complex_Programmable_Logic_Device.

[15] *Field Programmable Gate Array – Wikipedia*, [Online; accessed 26. May 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Field_Programmable_Gate_Array.

[16] *Allzweckeingabe/-ausgabe – Wikipedia*, [Online; accessed 7. Jun. 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Allzweckeingabe/-ausgabe.

[17] *Phasenregelschleife – Wikipedia*, [Online; accessed 6. Jun. 2018], Juni 2018. Adresse: https://de.wikipedia.org/wiki/Phasenregelschleife.

[18] *Verilog – Wikipedia*, [Online; accessed 26. May 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Verilog.

[19] S. Golson, *Oral History of Philip Raymond "Phil" Moorby*. Computer History Museum, 2013, S. 23–25. Adresse: http://archive.computerhistory.org/resources/access/text/2013/11/102746653-05-01-acc.pdf.

[20] *Very High Speed Integrated Circuit Hardware Description Language – Wikipedia*, [Online; accessed 26. May 2018], Mai 2018. Adresse: https://de.wikipedia.org/wiki/Very_High_Speed_Integrated_Circuit_Hardware_Description_Language.

A. Material

A.1 iCE40 PMOD-Pinbelegung

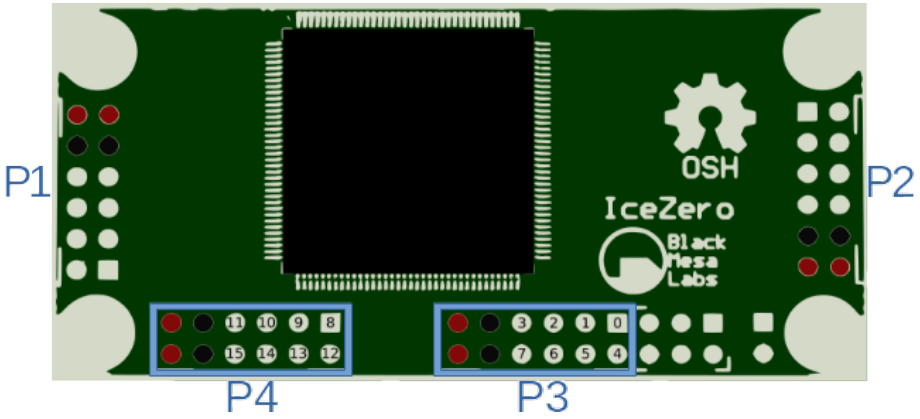

Abbildung A.1: Pinbelegung der PMOD-Header des icezero-Board (Quelle: Eigene Abbildung)

3.3V	GND	pin_11	pin_10	pin_9	pin_8
3.3V	GND	pin_15	pin_14	pin_13	pin_12

PMOD - P4

3.3V	GND	pin_3	pin_2	pin_1	pin_0
3.3V	GND	pin_7	pin_6	pin_5	pin_4

PMOD - P3

Tabelle A.1: Pinbelegung der PMOD-Header des Icezero-Boards

A.2 Pinverbindungen Raspberry Pi und FPGA-Shield

Tabelle A.2: Pinbelegung

ice40	WiringP	Name	Physical		Name	WiringPi	ice40
		3.3V	1	2	5V		
	8	SDA.1	3	4	5V		
	9	SCL.1	5	6	GND		
	7	1-Wire	7	8	TxD	15	
		GND	9	10	RxD	16	
	0	GPIO. 0	11	12	GPIO.1	1	
	2	GPIO. 2	13	14	GND		
	3	GPIO. 3	15	16	GPIO. 4	4	
		3.3V	17	18	GPIO. 5	5	
	12	MOSI	19	20	GND		
	13	MISO	21	22	GPIO. 6	6	
	14	SCLK	23	24	CE0	10	
		GND	25	26	CE1	11	
	30	SDA.0	27	28	SCL.0	31	
	21	GPIO.21	29	30	GND		
	22	GPIO.22	31	32	GPIO.26	26	
	23	GPIO.23	33	34	GND		
	24	GPIO.24	35	36	GPIO.27	27	
	25	GPIO.25	37	38	GPIO.28	28	
		GND	39	40	GPIO.29	29	

A.3 CD

B. GPL

Anhang B ...