Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InES Institute of
Embedded Systems

PA15_gelk_1 Polyphonic DDS Synthesizer mit MIDI Steuerung

Zürcher Hochschule für Angewandte Wissenschaften

Institute of Embedded Systems

Autoren	Katrin Bächli
Hauptbetreuer	
Nebenbetreuer	
Datum	14. Dezember 2015

Kontakt Adresse
c/o Inst. of Embedded Systems (InES)
Zürcher Hochschule für Angewandte Wissenschaften
Technikumstrasse 22
CH-8401 Winterthur

Tel.: +41 (0)58 934 75 25
Fax.: +41 (0)58 935 75 25

E-Mail: katrin.baechli@zhaw.ch
Homepage: `http://www.ines.zhaw.ch`

Liste der noch zu erledigenden Punkte

Zusammenfassung

Der erste Teil der Projektarbeit befasst sich mit zwei Herausforderungen der hardwarenahen Sprache VHDL.

Als Erstes werden sogenannte *glitches*, ungewollte Signalspitzen, künstlich herbeigeführt. Dies geschieht, in dem man die Signalpfade über externes Routing verlängert und dadurch der logische Wert verzögert beim nächsten Bautei, einem *decoder*, eintrifft. Wird der *decoder* asynchron betrieben, so verarbeitet dieser kurzzeitig falsche Werte, was in einem *glitch* sichtbar gemacht wird.

Als Zweites wird in einer Schaltung ein metastabiler Zustand provoziert. Damit dies erreicht wird, muss die *hold* - oder die *setup time* eines Flip-Flops verletzt werden. Erzeugt wird die Metastabilität durch unterschiedliches Takten zweier Logiken. Der Ausgang der ersten *clock domain* wird als asynchron Inpuls auf eine *finate state machine* einer anderen *clock domain* geführt. Die *finate state machine* fällt nach kürzester Zeit in einen undefinierten Zustand, einen Zustand, den sie nicht implementiert hat.

Der zweiten Teil der Projektarbeit beinhaltet das Entwickeln eines polyphonen *midi interface* für das Synthesizer-Projektes der Vorlesung Digitaltechnik II. Gemäss dem MIDI 1.0 Standard wird ein Controller implementiert, der auch das Drücken mehrer Tasten zuverlässig detektiert. Um die Entwicklung effizient zu gestalten, wird von Beginn weg mit einer textbasierten *testbench* gearbeitet. Für die Ausgabe der maxibal 10 gleichzeitig ertönenden Noten wird ein zweiter Block für das Händeln der ein und aus der einzelnen NOten geschrieben. Beide Blocks sind eingehend mit der testbench getestet und deren Verhalten gut dokumentiert.

(weglassen ??? Erst am Schluss nennen....) Als offener Punkt besteht die Implementation des *midi interfaces* in das bestehende Synthesizer-Projekt. Die Schnittstellen sind im Anhang festgehalten und die notewnigen Implementationsschritte, wie das Ausweiten des bestehenden DDS auf 10 DDS sind im Projekt als Blöcke eingebaut. Aus zeitlichen Gründen konnte dieser letzte Schritt nicht mehr während der Projektarbeit zu Ende gebracht werden.

Abstract

??? ou ou englisch.... mhhhmmmmmmmm

Vorwort

an Alexey: Bitte hier auf inhaltliche Richtigkeit prüfen. Brauche es erst am Do korrekt ...

Meine Motivation ist das vertiefte Kennenlernen der Sprache VHDL. Diese hardwarenahe Sprache beinhaltet mit der kombinatorischen Logik und der auch nicht sequentiellen Prozessverarbeitung Eigenheiten, mit denen ich Umgehen lernen wollte.

In der Projektarbeit waren es exakt diese Punkte, mit denen ich viel Zeit durch Debuggen verbrachte. Doch gerade so, ist mir nun diese Art der Programmierung vertrauter geworden und ich freue mich, auf kommende VHDL-Projekte.

Ich möchte Prof. Hans-Joachim Gelke meinen Dank aussprechen. Er legte den Fokus immer wieder auf die kombinatorische Logik und die Konsequenz des Codes, für das Umsetzen in der Hardware. Ebenfalls möchte ich Dr. Matthias Rosenthal danken, der diskret im Hintergrund die Arbeit mittrug und den Entwicklungsprozess mittrug.

Ich denke, dass diese Arbeit vor allem für Software Ingenieure interessant ist, da sie einen groben Einblick in die hardwarenahe Programmierung erhalten.

Inhaltsverzeichnis

1. Einleitung **7**
1.1. Ausgangslage . . . 7
1.2. Zielsetzung Aufgabenstellung Anforderungen . . . 7

2. Glitches **9**
2.1. Glitche in der Digitalen Signalverarbeitung . . . 9
2.2. Ursache für Glitches . . . 9
2.3. Glitches durch Pfadverzögerung . . . 10
2.4. Resultat . . . 12

3. Metastabilität **13**
3.1. Metastabiler Zustand . . . 13
3.2. Ursache von Metastabilität . . . 13
3.3. Metastabilität erzeugen . . . 14
3.3.1. Konzept . . . 14
3.3.2. Umsetzung . . . 15
3.4. Resultat Metastabilität provozieren . . . 15

4. MIDI Steuerung **20**
4.1. Einteilen der Blöcke und definieren der Schnittstelen . . . 20
4.2. Das MIDI Kommunikationsprotokoll . . . 21
4.2.1. *status bytes* . . . 21
4.2.2. *data bytes* . . . 21
4.2.3. Ungültige Bytes . . . 21
4.3. Umsetzung "midi controlBlock . . . 21
4.4. Umsetzung "polyphonie outBlock . . . 24

5. Polyphonie **25**
5.1. Midi Spezifikation . . . 26
5.2. Umsetzung . . . 26
5.2.1. software nahe . . . 26

6. testbench **27**
6.1. Textbasierte Testbench . . . 27
6.1.1. Struktur der Testserie . . . 27
6.2. code der testbench . . . 29
6.3. ergebnisse simulation . . . 29

7. Resultate der Projektarbeit **30**
7.1. Generieren von Glitches . . . 30
7.2. Zustand von Metastabilität provozieren . . . 30
7.3. MIDI Controller entwickeln . . . 30
7.4. Polyphonie Block . . . 30
7.5. DDS Generatoren basierend auf Frequenzmodulation entwickeln . . . 30
7.6. Textbasierte Testbench für alle entwickelten Blocks . . . 30

8. Diskussion und Ausblick **31**

9. Verzeichnis **32**
9.1. Literatur . . . 32

9.2. Glossar 32

A. Offizielle Aufgabenstellung **I**

B. Aufgabenspezifikation für den zweiten Teil **II**

C. Konzept Keyboard Dekoder **III**

D. Konzept Polyphonie **IV**

E. CD mit Projektdateien **V**

F. Top Synthesizer **VI**

G. In- und Outputdatei der textbasierte Testbench **VII**

1. Einleitung

Nennt bestehende Arbeiten zu diesem Thema (Literaturrecherche)

Stand der Technik: Bisherige Lösungen des Problems und deren Grenzen

1.1. Ausgangslage

Für den ersten Teil der Arbeit, die zwei ungewollten Effekte von *glitch* und einem metastabilen Zustand herzustellen gibt es selbsterklärend wenige Referenzprojekte. Beide Zustände sind nicht gewollt und finden als solche wohl oft Erwähnung in der Literatur [5] [6] [7], doch wie man diese Zustände provoziert, scheint bis auf eine gefundene [8], nicht von Interesse zu sein. Aus diesem Grund bestehen die ersten zwei Schritte vorwiegend aus eigenen Überlegungen, bzw. aus der Erfahrung von Prof. Hans-Joachim Gelke und seinen Anregungen.

Im zweiten Teil geht es um den Aufbau eines *midi interfaces*. MIDI bedeutet *musical instrument digital interface* und ist ein Standard, der sowohl die genaue Beschaffenheit der erforderlichen Hardware wie auch das Kommunikationsprotokoll der zu übermittelnden Daten festlegt [4]. Die MIDI Manufacturers Association dokumentiert die mehrfachen Erweitungen des MIDI 1.0 Standard [2]. Diese Spezifikationen sind relevant in der Entwicklung des Blocks *midi control*.

Am Institut for Embedded Systems bestand bereits die MIDI UART von Armin Weiss. Diese detektiert die empfangenen Bytes und sendet ein valid-Flag, wenn das Byte korrekt ist. Das Byte wird als logic Vetor übermittelt. In dieser Projektarbeit zu entwickeln sind deshalb die zwei Einheiten *midi control* und *polyphony out*. Und anschliessend diese Blocks in das bestehende Synthesizer-Projekt einzubauen.

Jeder zu entwickelnde Block wird mit einer textbasierten *testbench* getestet.

1.2. Zielsetzung Aufgabenstellung Anforderungen

Die offizielle Aufgabenstellung befindet sich im Anhang A unter refsect.aufgabenstellung. Alle Zitate beziehen sich aus diesem Text.

Von Anfang an war die Projektarbeit in zwei Teile geteilt:
Im ersten Teil sollten "Timing Artifakte demonstriert werden", die zu einem ßu einem vertieften Verständnis der digitalen Design Grundlagen führen.Ëin Ansatz, wie ein glitch detektiert und ein metastabiler Zustand aufgebaut werden kann ist gegeben:

- Ërzeugung von Glitches mit einem Zähler und nachgeschaltetem Dekoder. Sichtbarmachung der Glitches mit einem Oszilloskop. Betätigen des asynchronen Resets vom Decoder aus."
- "Provozieren und sichtbarmachung von Metastabilen Zuständen. Hierfür kann z.B. eine Schaltung mit zwei asynchronen externen Takten aufgebaut werden."

Der Fokus des zweiten Teils liegt im Projektausschrieb bei der Entwicklung vielfälltiger Klangfarben für das Synthesizer-Projekt:
Ïm zweiten Teil soll mit dem dem Direct Digital Synthesis Verfahren ein Synthesizer mit vielfältigen Klangfarben entwickelt werden. Damit kann anspruchsvolle digitale Schaltungstechnik umgesetzt werden. Zum erreichen der Klangvielfalt können mehrere DDS Generatoren gleichzeitig, mit unterschiedlichen Frequenzen und Phasen betrieben werden. Möglich ist auch eine Frequenzmodulation mit einem zweiten Generator oder Ändern des Volumens mit einer Hüllkurve.
Die Ansteuerung soll mit Hilfe eines MIDI Interfaces, welches Polyphonie (mehrere Klaviertasten gleichzeitig gedrückt) unterstützt. Die Implementierung soll im FPGA erfolgen. In der Implementierungsphase der Arbeit soll das Timing der FPGA Implementierung genau betrachtet werden.
Am Ende soll eine Referenzimplementierung in Anlehnung an den Yamaha DX7 für das Modul DTP2 entstehen."

Da die Entwicklung des ersten Teils länger dauerte, als vorausgedacht, wurde zu Beginn des zweiten Teils die neuen Anforderungen besprochen, da absehbar wurde, dass alle Anforderungen nicht realistisch sind (siehe Anhang B)

Gemäss der Spezifikation des zweiten Teiles sind die nächsten Schritte:

- "Midi Interface for Keyboard für Polyphonie nach Konzept von gelk
  o 10 Frequenz Control Ausgänge zur Steuerung der Tonhöhe des Generators
  o 10 On/Off Ausgänge Ton on/off
  o UART wird geliefert von gelk
  o VHDL wird von Grund auf neu erstellt.
- 10 DDS implementieren und mit Mischer Mischen
- Script basierte Testbench. Testbench erzeugt serielle Midi Daten, so wie sie auf dem DIN Stecker vorkommen (logisch)
- Testbench liest eine Testscript Datei ein, in welcher die Tastendrücke eines Keyboards abgebildet werden können. Midi Poliphony Spec muss durch die Testbench unterstützt werden können. Velocity muss nicht unterstützt werden."...
- "Kein VHDL code ohne Testbench.
- Block level testbench. Unit Tests."

Im Anhang C und D finden sich die vorgegebene Umsetzung des *midi interfaces*. Auf der CD befindet sich das Synthesizer-Referenz-Projekt, in welches das *midi interface* eingebaut wird.

2. Glitches

2.1. Glitche in der Digitalen Signalverarbeitung

In der Digitalen Signalverarbeitung ist glitch ein bekannter Fehler, den William I. Fletscher folgendermassen beschreibt: "Als *glitch* wird eine ungewollte, flüchtige "Signalspitze" bezeichnet, die Zähler aufwärts zählt, Register löscht oder einen ungewollten Prozess startet." [6]

Abbildung 2.1 zeigt zwei *glitches* in einem Ausgangssignal.

Abbildung 2.1.: Zwei Glitches im Ausgangssignal

2.2. Ursache für Glitches

Der Auslöser sind ungleichzeitig eintreffende Signale, die durch
1.) unterschiedlich lange Signalpfade,
2.) unterschiedliche Durchlaufverzögerungen der vorangehenden Flip-Flops oder
3.) unterschiedliche Logik-Zeiten
entstehen, und die in ein **asynchrones** Bauteil geführt werden. Der Dekoder im asynchronen Bauteil entschlüsselt dadurch kurzfristig einen falschen Wert.

Abbilung 2.2 zeigt ein leicht verzögertes (getaktetes) enable-Signal zu einem anders verzögerten (getakteten) Flip-Flop-Eingangssignal Q. Der Ausgang des Flip-Flops weist kurzzeitig Glitches auf.

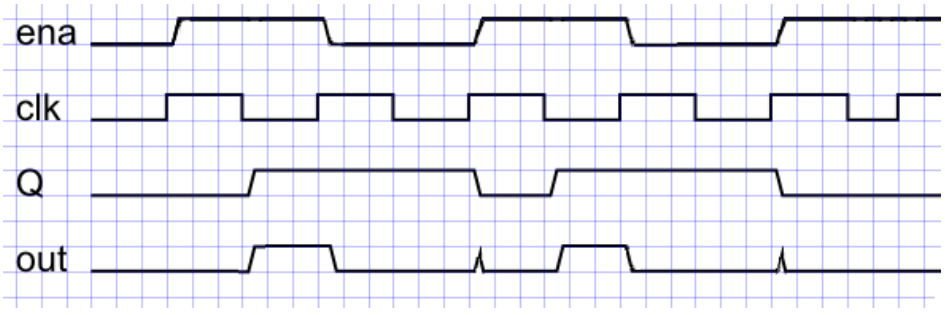

Abbildung 2.2.: Asynchrone Eingangssignale führen zu Glitches

2.3. Glitches durch Pfadverzögerung

Konzept

Ein asynchroner Zähler erhält verzögerte Bitwerte. Zählt man binär auf 15, so kann sich beim Übergang von der Zahl 11 zu 12, die falsche Zahl 15, ergeben, sofern die zwei höheren Bits der Zahl 11 verzögert ankommen (siehe Abbildung 2.3).

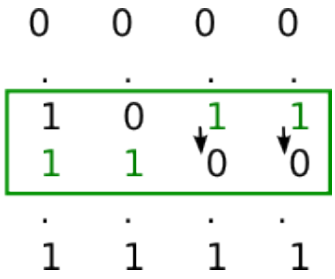

Abbildung 2.3.: Binärwerte des asynchronen Zählers

Die Verzögerung der zwei Bits, wird über Routing umgesetzt.

Implementation

Die Hardware ist das altera board De2 mit dem FPGA Cyclone II. Kompiliert wird das Projekt mit Quartus 13.0sp, der ältsten Quartus-Version, die den Cyclone II unterstützt.

Die Pfad*verlängerung* wird über das Routing über die GPIO-Pins des Headers 1 gemacht (siehe Abbildung 2.6. Dekodiert die asynchrone Logik die Zahl 15, wird das Reset-Signal an den Zähler gesendet und der Zähler beginnt wieder von 0 an zu zählen. Produziert der Dekoder zur falschen Zeit einen Reset, so ist dies eine Fehlkodierung: ein *glitch*.

Das RTL-Diagramm des asynchronen Zählers sieht wiefolgt aus:

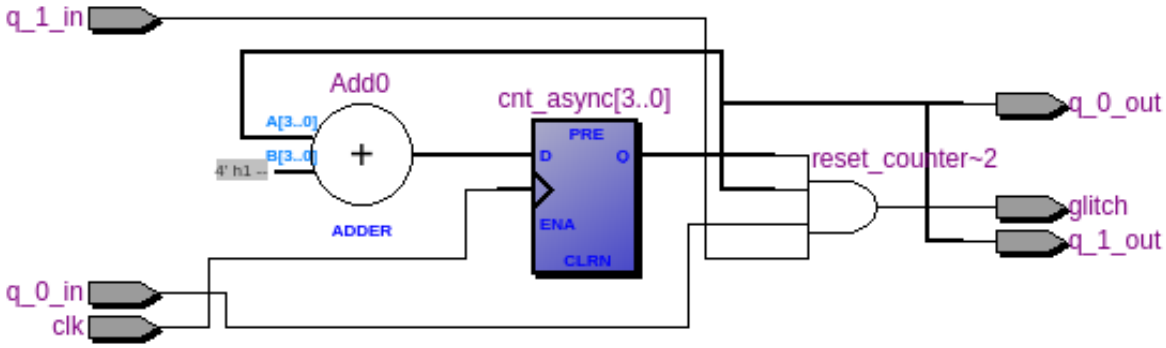

Abbildung 2.4.: Asynchroner Zähler mit Routing erzeugt Glitch

Um die Lösung gegen *glitches* aufzuzeigen, wird dem asynchronen Zähler zur Synchronisation ein Flip-Flop nachgeschalten. Dadurch werden die asychronen Zustände übersehen.

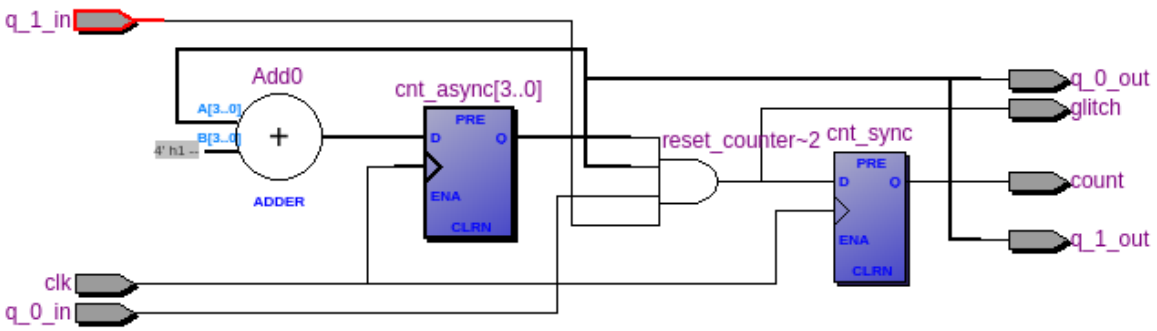

Abbildung 2.5.: Glitch-Zähler und synchroner Zähler dazu

Die Reset-Signale des asynchronen Zählers wie die des synchronisierten Zählers werden an die GPIO Headers ausgegeben, ebenso der Systemtakt.In der Abbildung 2.6) wird das Signal des asynchronen Zählers als Glitch und das Signal des synchronisierten Zählers als Count benannt. In der GPIO-Pinbelegung sieht man auch die Nutzung der zwei oberen Pin-Reihen für das Routing (benannt mit Routing OUT, IN). Der Systemtakt wird als CLK ausgegeben.

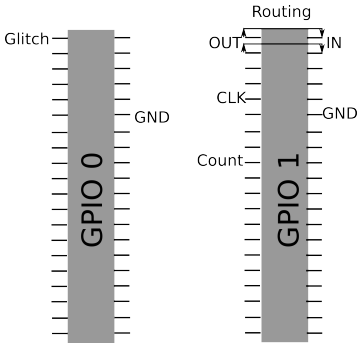

Abbildung 2.6.: GPIO Anschlüsse

2.4. Resultat

Der Reset des asynchronen Zählers (CH 1), der synchronisierte Reset (CH 2) und der Systemtakt (CH 3) werden am KO ausgegeben. Durch die Synchronisation wird der Wert um 1 Periode (= 20 ns) verzögert.

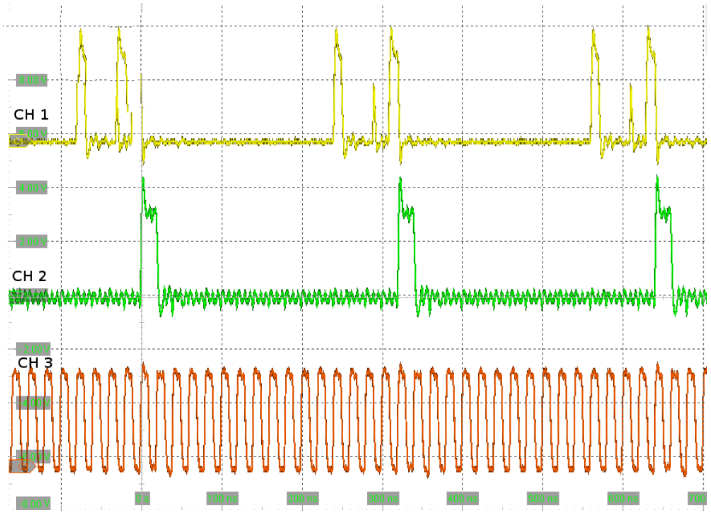

Abbildung 2.7.: Glitch (gelb), Zähler (grün) und Takt (orange)

Das *glitch* trifft in der im Übergang von der 11 zur 12 Periode (= 240 ns) regelmässig auf. Dies ist das zu erwartende Ergebnis. Ein kurzzeitiges asynchrones Verhalten findet sich auch im Übergang von der 13 zur 14 Periode. Dies ist wenn der binäre Wert 1101 auf 1101 wechselt. Da die zwei niederwertigen Bits verzögert sind, ist das dekodierten des Wertes 1111 plausibel.

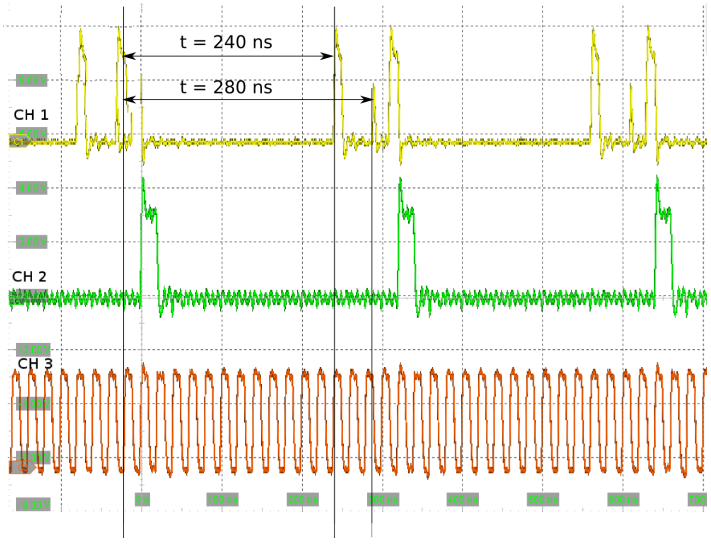

Abbildung 2.8.: Zeitanalyse Glitches

3. Metastabilität

3.1. Metastabiler Zustand

Metastabilität bedeutet, dass der Ausgang eines Flip-Flops nicht dem Eingang entsprechen *muss*. In einem metastabilen Zustand kann ein Ausgang korrekt sein, muss aber nicht. Im Idealfall wählt wählt ein Flip-Flop seinen Ausgangswert selbst (siehe Abbildung **??** oberes Signal). Im schlechten Fall "hängt" sich das Flip-Flop "auf" und toggelt permanent zwischen '0' und '1' (Abbildung **??** unteres Signal).

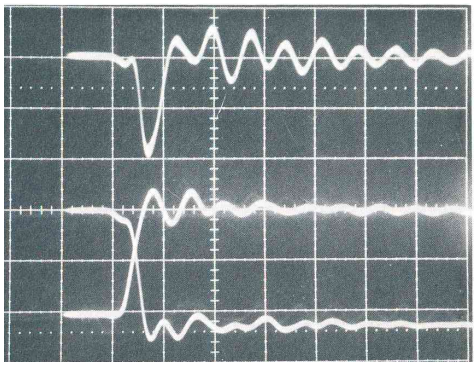

Abbildung 3.1.: Metastabilität schlimmster Fall [7]

3.2. Ursache von Metastabilität

Die Ursache unsicherer Ausgangswerte liegen darin, dass das Inputsignal eines Flip-Flops zur falschen Zeit wechselt.

"If data inputs to a flip-flop are changing at the instant of the clock pulse, a problem known as *metastability* may occur. In the metastable case, the flip-flop does not settle in to a stable state" [5]

"If the amplitude of the runt pulse is *exactly the treshold level of the SET input of the output cell*, the cell will be driven to its metastable state. The metastable state is the condition that is roughly defined as "half SET and half RESET" [7]

Trifft der anzulegende Wert zu spät ein wird die *setup time*) verletzt und wird der Signalwert zu früh entwendet, verletzt die *hold time*). Metastabilität kann vermieden werden, wenn diese zwei Zeiten strikt eingehalten werden:

"Metastabilit is avoided by holding the information stable before and after the clock pulse for a set period of time, called the setup time for the data line an the hold time for the control line." [5]

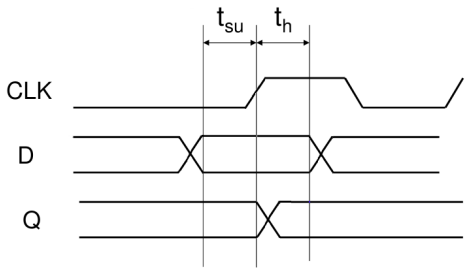

Abbildung 3.2.: Einhalten der Datenzeiten

Um Metastabilität zu vermeiden, sollte die Logik möglichst klein, die Bauteile beieinander und der Systemtakt an die längste Pfadzeit angepasst werden. Der maximal erlaubte Systemtakt kann in quartus mit dem Timequest Time Analyser abgefragt werden.

3.3. Metastabilität erzeugen

3.3.1. Konzept

Aufgebaut wird ein System mit zwei *clock domains*. Eine *clock domain*, Gebiet 1, beinhaltet einen Zähler, der an das Gebiet 2 asynchrone Impulse sendet. Gebiet 2 verarbeitet diese Impulse in einer *finate state machine*. Bei korrekter Funktionsweise wechset die *fsm* zwischen den definierten *states*. Funktioniert sie falsch, fällt die *fsm* in einen *state*, den sie nicht implementiert hat.

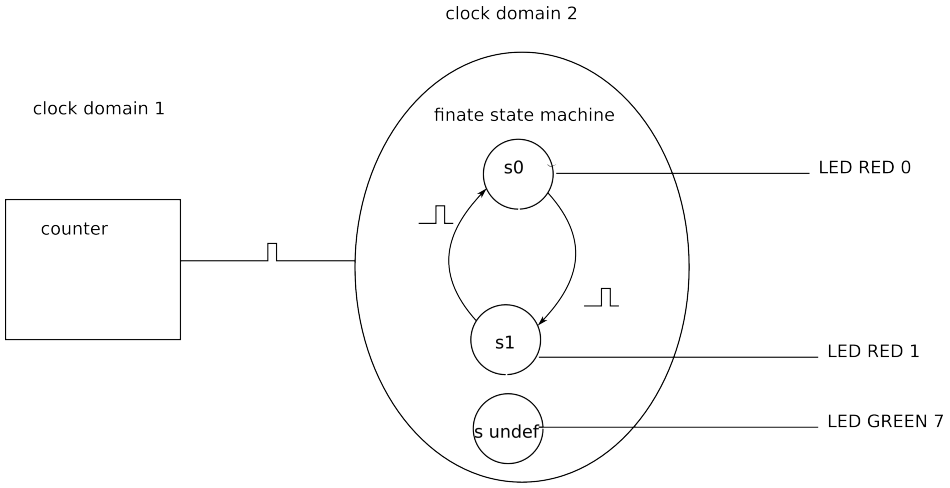

Abbildung 3.3.: Konzept Metastabilität nachweisen

3.3.2. Umsetzung

Als Hardware wird das altera development board De2 genommen und mit der Software quartus 13osp0 gearbeitet. Die die zwei Takte nicht Vielfache voneinander sein dürfen, wure für den Zähler ein Takt von 27 MHz und für die *fsm* ein Takt von 50 MHz. Der Takt des Zählers ist leicht schneller als die Hälfte der fsm und schiebt sich vorwärts (siehe Abbildung 3.4). Das Verletzen der *setup time* ist eine Frage der Zeit.

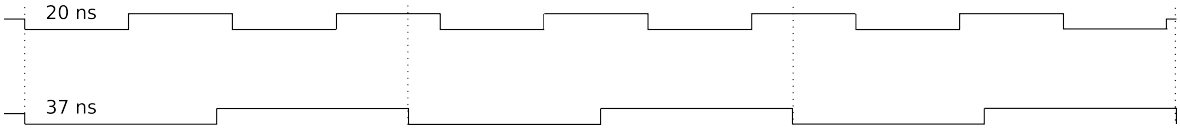

Abbildung 3.4.: Die zwei Taktzeiten

Die Zustandsüberprüfung erfolgt über das Ausgeben des aktuellen Zustands auf den zwei roten LEDs.

- Zustand = s0
  Rote LED 0 ist an
- Zustand = s1
  Rote LED 1 ist an
- Zustand = *others*
  Grüne LED 17 ist an

Funktioniert die *fsm*, blinken die zwei roten LEDs abwechslungsweise. Fällt die *fsm* in einen undefinierten Zustand, leuchtet die grüne LED. Um die Ursache der Metastabilität, das Verletzten der *setup time* zu verhindern, wird eine optionale Synchronisation durch Switch 17 eingebaut. Ist Swicht 17 auf '1', wird der Puls der *clock domain* 27 MHz durch ein Flip-Flop auf 50 MHz synchronisiert.

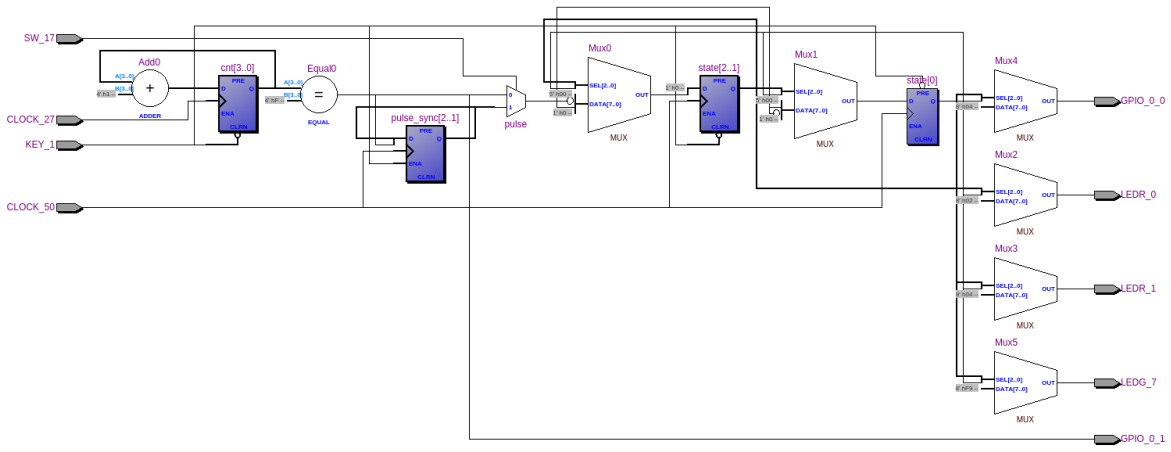

Abbildung 3.5.: RTL mit Synchronisations-Switch

3.4. Resultat Metastabilität provozieren

Das Resultat ist, dass das Board unmittelbar nach Einstellen in den metastabilen Zustand fällt und die grüne LED leuchtet. Wird Reset gedrückt, folgt ein kurzes Aufblinken der zwei roten LEDs und

wieder die grüne LED.

Abbildung 3.6.: Metastbiler Zustand

Wird die Synchronisations-Schaltung betätigt, leuchten beide roten LEDs auf. Die *fsm* wechselt zwischen den states s0 und s1 hin und her. Das Verbleiben in den zwei definierten Zuständen s0 und s1 funktioniert auch nach einem Tag noch.

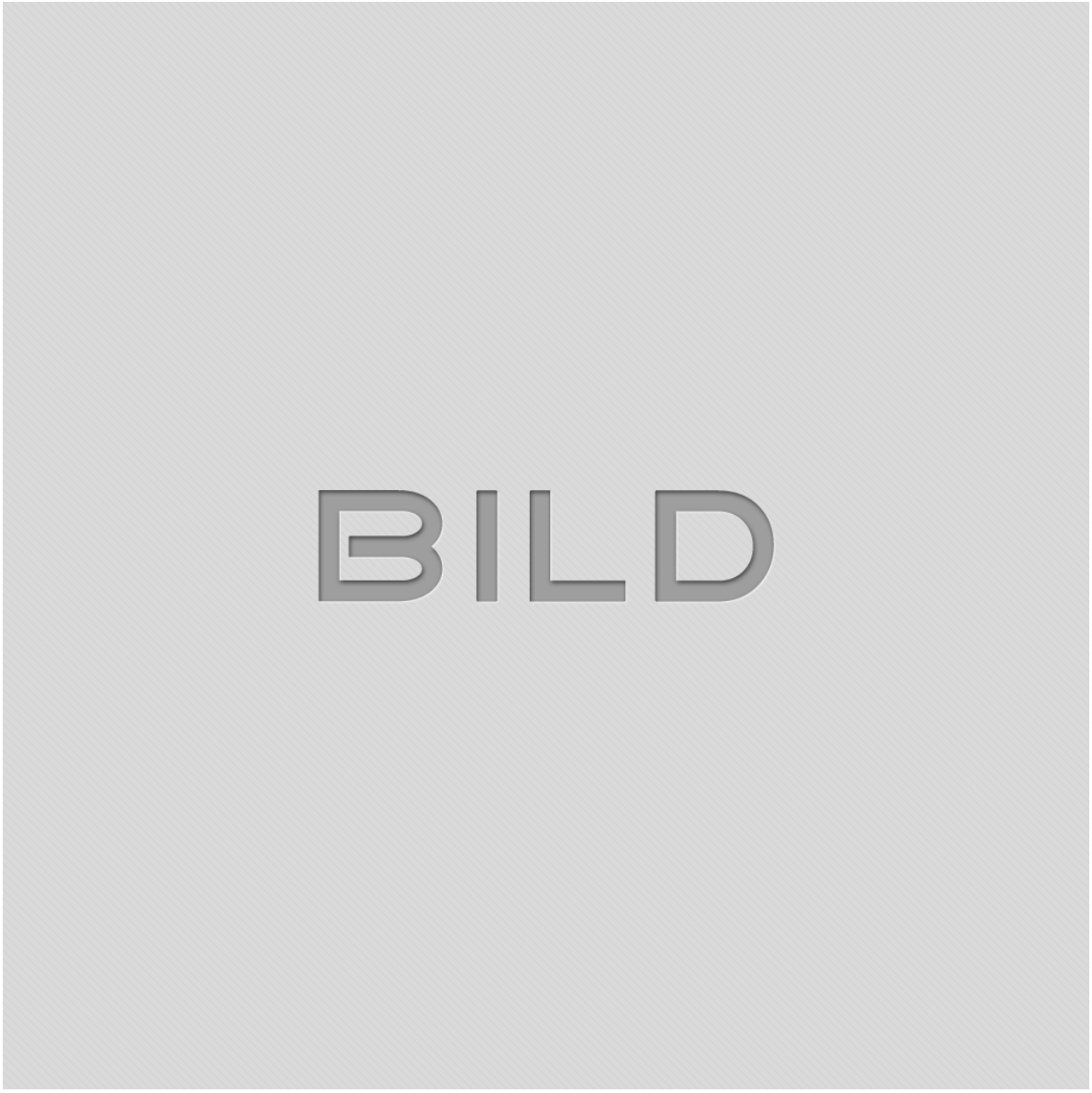

Abbildung 3.7.: Switchschalter ON: Rote LEDs leuchten

Wird das Wechseln zwischen den zwei states am KO ausgegeben, so erkennt man, da - weil der Takt 27 MHz kein Bruchteil von 50 Mhz - kein wiederkehrendes Muster der Wechsel zwischen den zwei Zuständen auftritt.
CH 1 = Rote LED CH 2 = Synchronisierter Puls

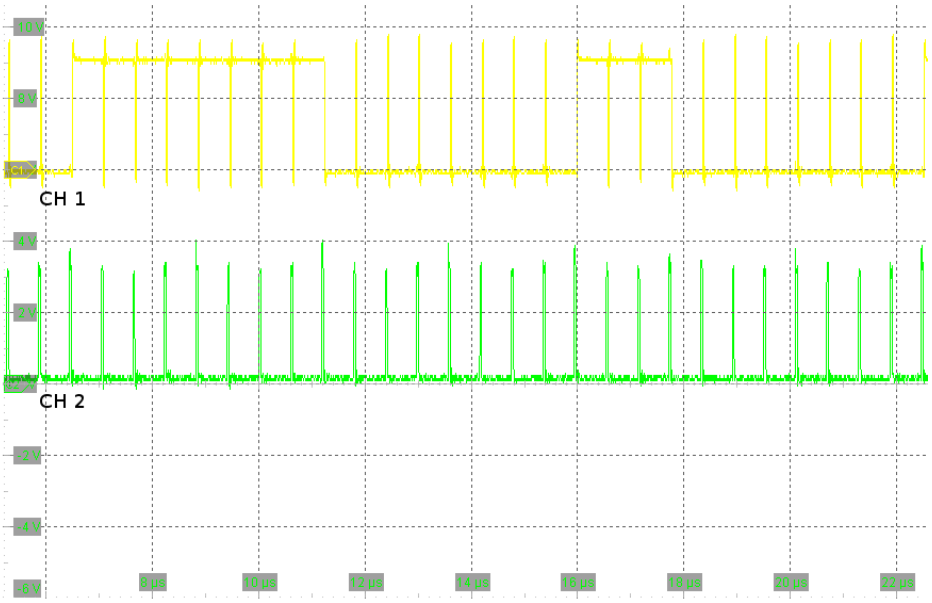

Abbildung 3.8.: Unregelmässiger Wechsel zwischen Zustand s0 und Zustand s1

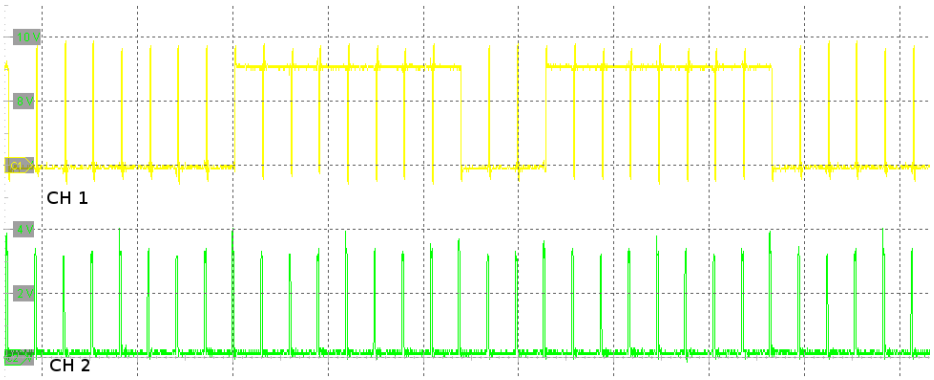

Abbildung 3.9.: Unregelmässiger Wechsel zwischen Zustand s0 und Zustand s1

Im Zustand der Metastabilität sind die Pulse nicht synchronisier und die rote LED geht nicht an.

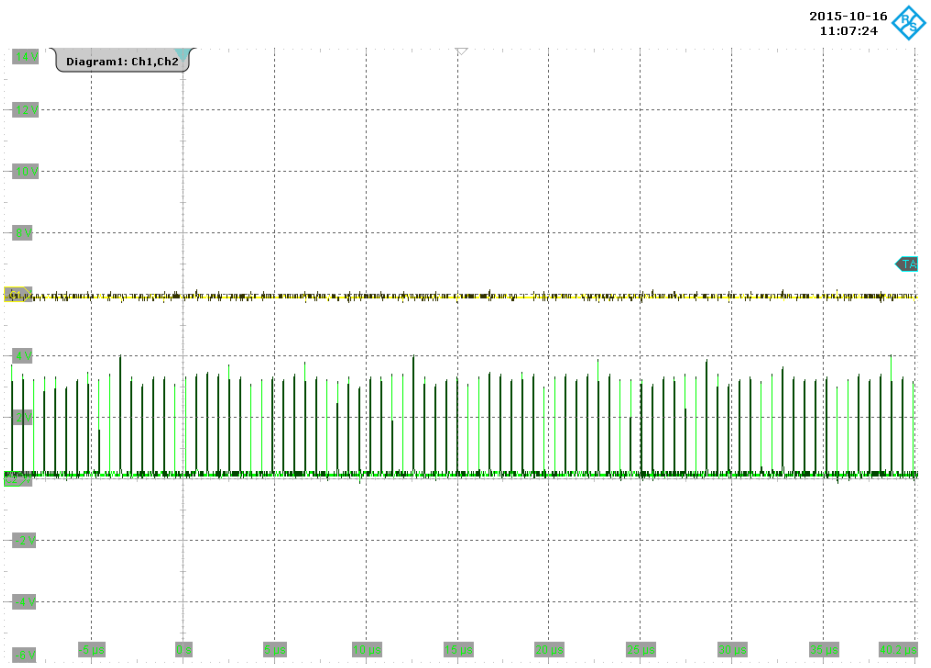

Abbildung 3.10.: Metastabiler Zustand

4. MIDI Steuerung

4.1. Einteilen der Blöcke und definieren der Schnittstelen

Beschreiben der Schnittstellen
Vor Programmieren der units (blöcke) und unit test, schnittstellen klären
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Die Abbildung F.1 (im Anhang F) zeigt, wie das zu entwickelnde MIDI Interface in die bestehenden Blöcke des Synthesizer-Projektes eingebaut wird. Die im Anhang direkt anschliessende Abbildung F.2 zeigt dann die geplante Umsetzung detaillierter.

Im folgenden wird nur auf den Block *midi interface* eingegangen, der die Umsetzung der MIDI Steuerung darstellt. Als erstes die Zusammenfassung der internen Blöcke (siehe Abbildung 4.1).

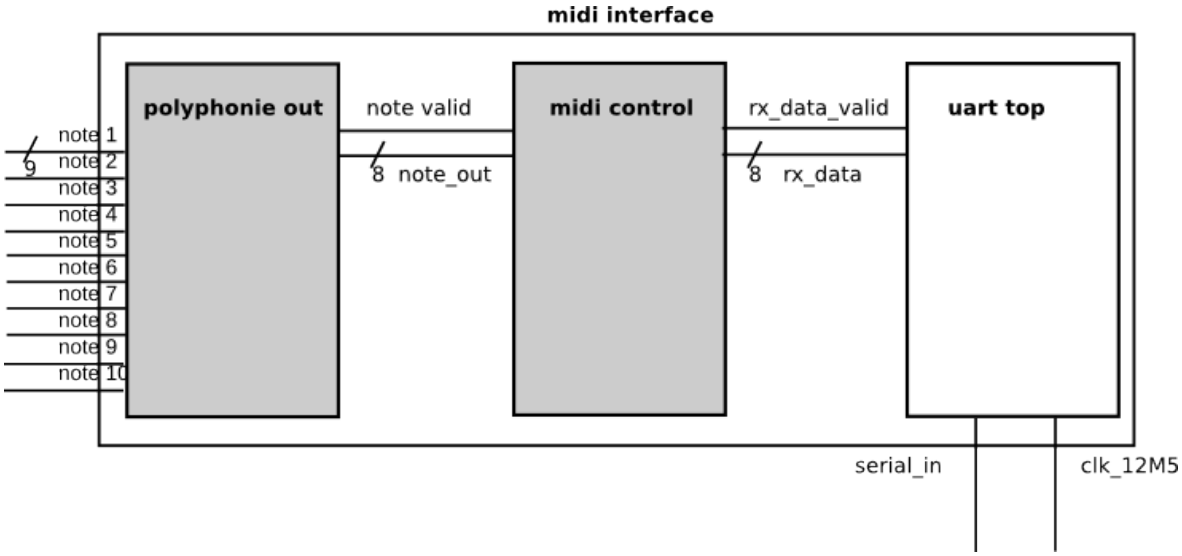

Abbildung 4.1.: Blockschaltbild MIDI Interface

Gegeben ist der UART Block, in der Abbildung als uart top bezeichnet. Dieser detektiert MIDI Bytes und sagt, ob diese gültig sind. Zu entwickeln sind die zwei Blöcke "**midi control**ünd "**polyphonie out**".

Als erstes wird kurz generall auf das MIDI Kommunikationsprotokoll eingegegangen, damit man die Kriterien bei der Umsetzung versteht. Danach wird die Umsetzung des *midi control*-Blocks erklärt und zum Schluss die Implementation des *polyphone out*-Blocks.

4.2. Das MIDI Kommunikationsprotokoll

Werden MIDI Daten übermittelt, so unterscheidet der Standard zwei Typen an Daten (Detailierte Spezifikation S. 11 - 12).

4.2.1. status bytes

status bytes sind 8 Bit lang und das MSB ist immer logisch '1'. Die Status Bytes dienen dem Identifizerein der nachfolgenden *data bytes*. Das *status byte* sagt, von welcher Art und mit welcher Datenstruktur die folgenden *data bytes* sein werden.
MIDI behält einen Status so lange, bis ein neues *status byte* folgt. Dieses Verhalten wird als *running status* bezeichnet. Dieses Verhalten ist vor allem für die Polyphonie interessant, da dieser Zustand bleibt und viele *Data Bytes* (im Sinn von Noten) folgen können, ohne dass es eines neuen Status Bytes bedarf.

4.2.2. data bytes

Gemäss Spezifikation folgen einem *Status Byte* exakt ein oder zwei Bytes. Das MSB ist immer logisch '0'. Die Werte können von 0x00 bis 0x7F sein. Das bedeutet, dass MIDI maximal 128 Noten unterscheiden kann.
data bytes sind Noten, Geschwindigkeit des Anschalges und ...
Je nachdem welches *status byte* im Voraus gesetzt wurde, werden die Attribute anders interpretiert. Ist z.B. Polyphonie gesetzt, so bedeutet ein *data byte* mit einer Geschwindigkeit von 0, Note abstellen. Diese und andere Spezifikationen werden detailliert in **Detailierte Spezifikation** beschrieben.
Ëmpfänger sollen so konzipiert sein, dass zuerst alle*data bytes* empfangen werden und ein neues *status byte* kommt. Danach werden ungültige Daten verworfen. Einzige Ausnahme ist der *running status*. Bei dem nicht bis zum Ende gewartet wird."(Spezifikation, S. 6).

4.2.3. Ungültige Bytes

Älle *status bytes*, die nicht implementierte Funktionen enthalten und alle ihnen folgenden *Data Byte*s sollen vom Empfänger verworfen werden."**(Spezifikation, 6)**.
MIDI Geräte sollen ausdrücklich beim Ein- und Abstellen darauf bedacht sein, dass keine undefinierten Bytes gesendet werden **(ebd)**.
Diese Anforderung ist wichtig beim Implementieren einer Finate State Machine und der Testbench (**siehe Kapitel .....**).

4.3. Umsetzung "midi controlBlock

Ausgehend von der Spezifikation sind drei Eckpunkte für die *finite state machine* zu berücksichtigen:

1. Unterscheiden von *status byte* und *data byte*
2. Unterschiedliche Interpretation der *data bytes* abhängig vom *status byte*.
3. Verwerfen aller falschen *status byte* oder *data bytes*

Zu beachten in der Verarbeitung der Daten ist, dass im *single mode* zuerst gesagt wird, ob eine Note an oder ab und diese Reihenfolge im *polyphony mode* gerade umgekehrt ist: zuerst kommt die Note, dann die Angabe, ob sie an oder ab ist.

Aus diesen Anforderungen ergab sich folgende *finite state machine*:

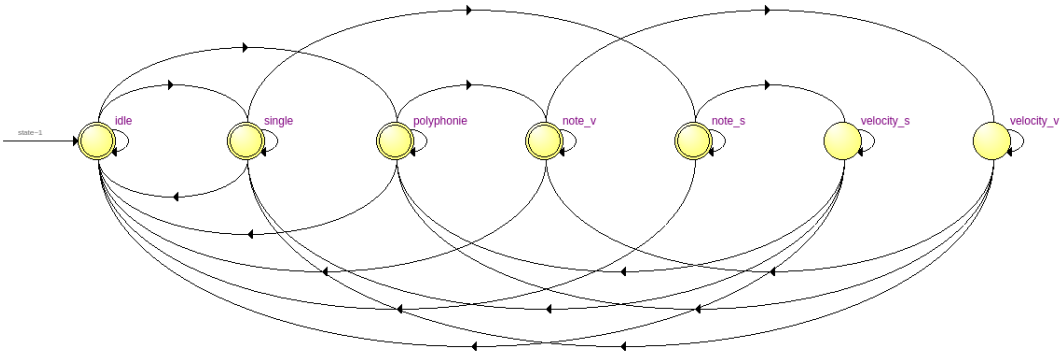

Abbildung 4.2.: fsm Übergänge

Alle drei Anforderungen sind implementiert:
Vor jedem *data byte* muss zuerst ein *status byte* eingegangen sein. Die *finite state machine* fragt im *idle* Zustand nur nach den *status bytes*. So weisen die oberen Nibbel mit den Werten "1001"für NOTE AN und "1000"für NOTE AUS auf den *single mode* hin und der Wert "1010äuf den *polyphony mode*. Nach diesen *status bytes* erwartet die *finate state machine data bytes*.
Die unterschiedliche Reihenfolge von Noten-Byte und Angaben zu an oder ab, wurde statusabhängig umgesetzt:
Im *single mode* wird das vierte Bit des *status nibbel* zum Setzen von AN und AB verwendet . Im polyphony mode wird das zweite *data byte*, das ueblicherweise die Geschwindigkeit der Note bestimmt, für das Setzen von AN und AUS genommen. Ist der Wert der Geschwindigkeit gleich NULL, dann soll in diesem Zustand die Note als AUS gelten.
Gut sichtbar ist, dass die *finite state machine* bei ungültigen *bytes*, die Daten verwirft und in den *idle* Zustand zurückgeht.

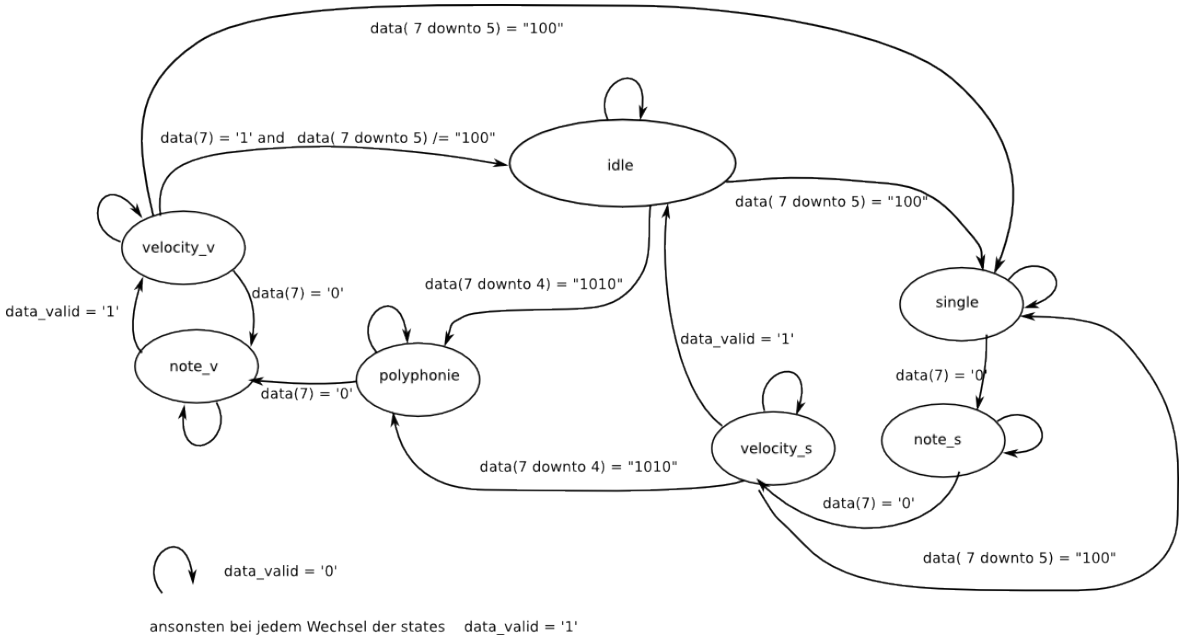

Abbildung 4.3.: fsm Übergangsbedingungen

Das Umsetzung der *fsm* sieht man in der simulation in den zweit unteren Abbildungen gut. Nach dem Zustand *idle* folgt das *status byte* ßingle"(Abbildung 4.4)oder "polyphonie"(Abbildung 4.5)

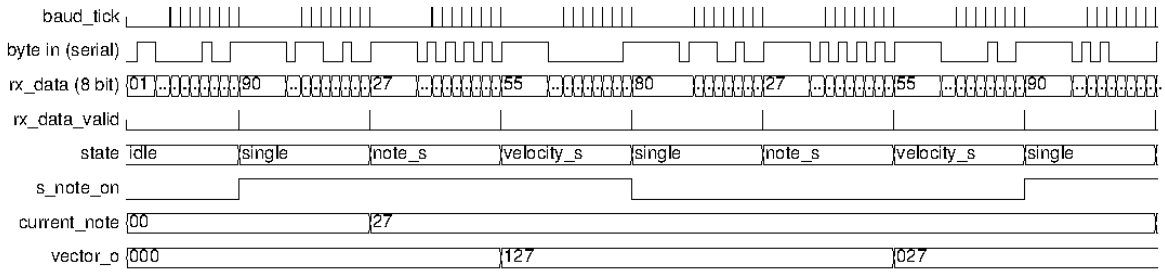

Abbildung 4.4.: fsm für single mode

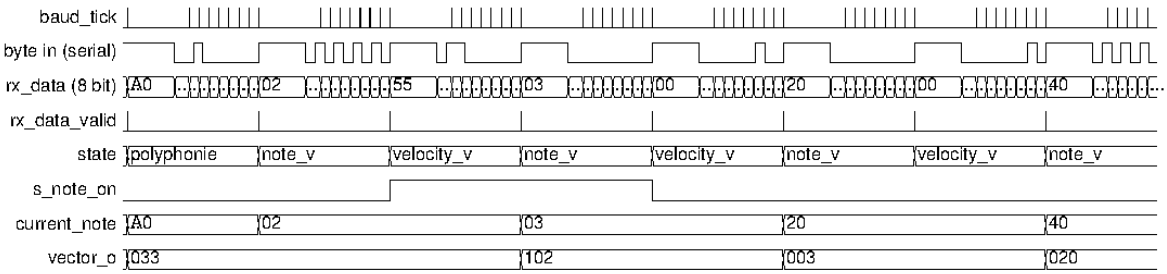

Abbildung 4.5.: fsm im polyphony mode

Die Simulation wurde mit möglichen Fehlübertragungen angereichert, damit man sieht, ob der *midi control* weiter korrekt funktionert. Eine detaillierte Beschreibung zum Aufbau der Simulation befindet sich im Kapitel **testbench**.

4.4. Umsetzung "polyphonie outBlock

5. Polyphonie

Abbildung 5.1.: Bildbeschreibung ....

5.1. Midi Spezifikation

5.2. Umsetzung

5.2.1. software nahe

hardware nahe

6. testbench

Inspiriert vom Konzept des *test driven development* wird stets parallel zur Entwicklung einer *unit* (im folgenden als Block genannt) der *unit-test* entwickelt [3].

Nachdem im Voraus die Schnittstellen zwischen den Blocks geklärt sind (**siehe Kapitel xXX**) wird eine leere Hülle für jeden zu entwickelnden Block erstellt. Die *testbench* geht von Beginn weg vom Ziel, eines funktionstüchtigen *midi interface* aus. Solange ein Block noch nicht fertig entwickelt ist, führend erst partiell korrekt führende Signale an den Ausgang der DUT. Mit jeder zusätzlichen Implementation, wird das Verhalten der Signale korrekter.

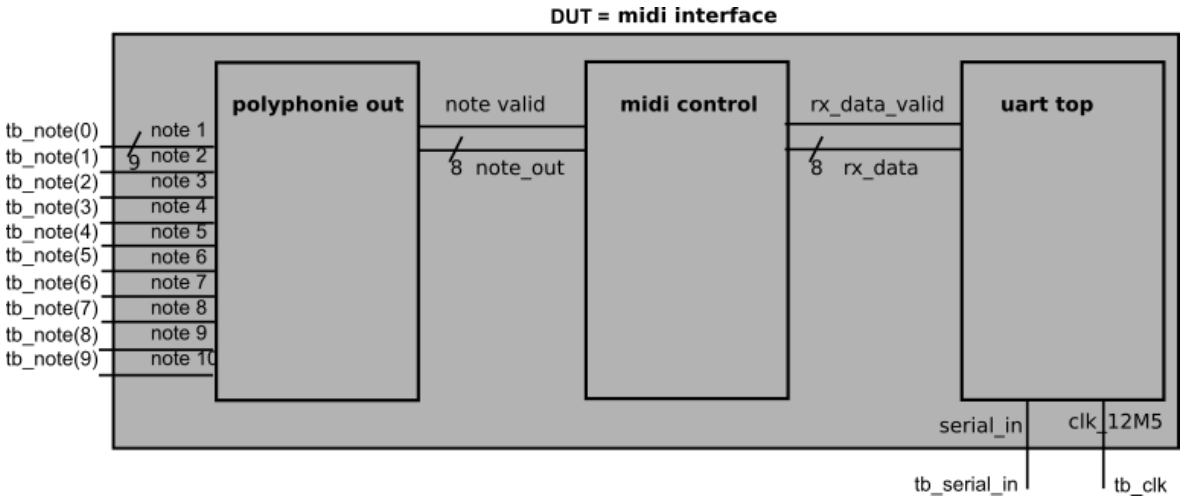

Abbildung 6.1.: Blockschaltbild Device under Test

Eine Bedingung an die Projektarbeit ist, dass die Tests textbasiert ablaufen sollen (siehe Aufgabenstellung im Anhang B). Diese zusätzliche Spezifikation wird hier kurz zusammengefasst.

6.1. Textbasierte Testbench

Die zu testenden Befehle stammen aus der Datei input_midi.txt. Der Inhalt ist aufgelisten im Anhang G.

6.1.1. Struktur der Testserie

Ausgehend von der MIDI 1.0 Spezifikation, wird die Funktion des *single modes* und des *polyphony modes* getestet. Der Testablauf hängt von dem zu testenden Fall ab: Weil die *data bytes* im *polyphony mode* anders sind, als im *single mode*, baut sich der Test auch anders auf.

Setzen des Testmodus

Damit das Programm weiss, nach welchem Aufbau es die Signale senden muss, wird zuerst der Testmode eingestellt. Dies ist das erste Wort jeder Zeile im Testscript.

reset 00 00 ....
check00 00 ....
singl 55 90 ....

Den Testmodus **reset** braucht es, da keine manuellen Tests in die *testbench* eingeführt werden. Den Testmodus **check** beinhaltet die zu erwartenden Ergebniswerte, wenn der vorhin gesetzte Testmode korrekt abläuft.
Da das Testmodus Kommando-Wort immer gleich lang sein muss (hier 5 Buchstaben) sind die Wörter grammatikalisch nicht korrekt. Aus polyphony wurde polyp.

Der Tokenaufbau der zwei Testmodes *single mode* und *polyphony mode* wird nun genauer erklärt:

Einzelne Noten testen

Das Testen einer einzelnen Note ist einfach. Da gemäss Spezifikation zuerst ein *status byte* mit der Meldung Note on (0x90) oder Note off (0x80) kommt. Und dann das *data byte* mit dem Notenwert folgt. Die Geschwindigkeit ist für das An- oder Abstellen der Note nicht relevant und wird deshalb nicht als Befehl eingelesen. Die *testbench* hängt selbständig im single mode nach jeder Note einen Dummy-Geschwindigkeitswert von 0x55 an.

Zeile in der Datei

singl	55	90	27	80	27	90	05	00	00
check	00	00	27	00	00	00	05	00	00

Tokenaufbau

mode	Note	Velocity	Note	Velocity	Note	Velocity	Note	Velocity	Anzahl Noten an

Die hier abgebildete Sequenz bedeutet: Note 27 anstellen (0x90 ist das status byte für Note an), dann Abstellen der Note 27 und am Schluss Anstellen der Note 05.
Überprüft am Schluss der Sequenz die *testbench* den Ausgang des *midi interfaces*, so erscheinen die zwei Notenwerte 27 und 05.

Im Fall der einzelnen Note passt die Tokenstruktur, die sich am schwierigsten Fall, der Polyphonie orientierte, nicht ganz. Im Gegensatz zur Polyphonie muss bei der einzlenen Note VOR dem Notenwert ein status byte kommen. Damit dies in der Tokenstruktur umsetzbar ist, wird zuerst ein Dummy-Wert (55) zum Verwerfen der *testbench* übergeben. Erst dann folgt das *status byte* und dann, analog zur Polyphony-Sturktur folgt die erste Note. Im Nachhinein erscheint mir dieser Aufbau als zu kompliziert und ich würde ein nächstes Mal mehr mode-spezifisch die Datenauswertung gestalten.

Polyphonie testen

Zeile in der Datei

polyp	71	55	02	55	33	55	08	00	00
check	71	00	02	00	33	00	00	00	03

Tokenaufbau

mode	Note	Velocity	Note	Velocity	Note	Velocity	Note	Velocity	Anzahl Noten an

Das bedeutet, dass die *testbench* im Testmodus Polyphonie ist. Und deshalb als erstes das *status byte* der Polyphonie "10100000"(0xA0) senden muss. Danach werden die Token gemäss der Polyphonie-Spezifikation interpretiert: Die *data bytes* wechseln sich ständig mit dem Notenwert und der dazugehörenden Geschwindigkeit ab. Hier in der Testdatei hält die erste Note den Wert 71 und wird gefolgt von irgendeiner Geschwindigkeit (hier Dummy-Wert 55), usw.. Kommt ein Geschwindigkeitswert von 0, so bedeutet dies, dass die Note abgestellt wird. In der Polyphonie ist das Note An- und Abstellen asynchron. Der Befehl folgt nicht automatisch nach jedem Notenbyte. Aus diesem Grund weiss man nicht indirekt, wie viele Noten zu einem gewissen Zeitpunkt an sind. Aus diesem Grund gibt es den letzten Token: Anzahl Noten an. Dieser sagt der testbench wie viele Noten an sein sollen. Beim Überprüfen des Ausganges des *midi interfaces* muss nach der ersten Befehlzeile die drei Noten 71, 02 und 33 an sein. Was einer Summe von drei entspricht.

6.2. code der testbench

Im Gegensatz zum hardwarenahen Code der VHDL-Blocks, bei denen arrays und loop explizit vermieden wurden, baute die *testbench* bewusst auf softwarenahe Strukturen auf.

package

Es ist ein Package für die Konstanten der der *status bytes* und für die Noten-Verarbeitung erstellt.

Die Tokenstruktur wurde wiefolgt implementiert:

```
– define midi_data
type t_midi_data is record
        token_note : std_logic_vector(7 downto 0);
        token_attribut : std_logic_vector(7 downto 0);
        end record;


type t_midi_data_array is array (0 to 3) of t_midi_data;


– define token structure
type t_token_line is record
        token_cmd : string(1 to 5);
        t_midi_data : t_midi_data_array;
        token_number : std_logic_vector(7 downto 0);
        end record;


– array with note structure (input/output)
type t_note_array is array (0 to 9) of std_logic_vector(8 downto 0);
```

Prozessstruktur

Durch das Aufteilen der Prozesse ergab sich auch die Unterscheidung zwischen lokalen Variablen und Signalen.

6.3. ergebnisse simulation

7. Resultate der Projektarbeit

Zusammenfassung der Resultate

7.1. Generieren von Glitches

Beides erreicht. Viel Aufwand, da wenig Wissen wie ungewollter Zustand erzeugt werden kann. Es dauerte 4 Wochen (15. oktober + Doku), der insgesammt 16 Wochen PA.

7.2. Zustand von Metastabilität provozieren

Beides erreicht. Viel Aufwand, da wenig Wissen wie ungewollter Zustand erzeugt werden kann. Es dauerte 4 Wochen (15. oktober + Doku), der insgesammt 16 Wochen PA.

7.3. MIDI Controller entwickeln

7.4. Polyphonie Block

Midiansteuerung nach vielen Redesignes gelungen. Mehr in der Software geübt, ist das Timing in VHDL übungsbedürftig.

7.5. DDS Generatoren basierend auf Frequenzmodulation entwickeln

Sitzung: Nur 10 DDS einbauen. Schnittstellen da. Nicht da,Frequenzmodulation anstelle von LUT. Aus zeitgründen. Nur erster Entwurf, wie es umzusetzen ist.

7.6. Textbasierte Testbench für alle entwickelten Blocks

8. Diskussion und Ausblick

Bespricht die erzielten Ergebnisse bezüglich ihrer ERwartbarkeit, Aussagekraft und Relevanz
Interpretation und Validierung der Resultate
Rückblick auf Aufgabenstellung: erreicht nicht erreicht

Legt dar, wie die Resultate weiterhin genutzt werden können
an sie angeschlossen werden kann

9. Verzeichnis

9.1. Literatur

[1] Altera. *Cyclone IV Device Handbook*, pages 8–19. San Jose, 2014.

[2] The MIDI Manufacturers Association. *MIDI 1.0 Detailed Specification.* Los Angeles, 1995.

[3] Kent Beck. *Test-Driven Development, By Example*, page ix. Addsion Wesley Signature, 2013.

[4] Christian Braut. *Das MIDI-Buch*, page 10. Sybex, 1993.

[5] John A. Camara. *Engineering Reference Manual*, pages 32–2. Belmont, 2010. About metastability.

[6] William I. Fletcher. *An Engineering Approach to Digital Design*, page 472. Utah State University, 1980. About glitch.

[7] William I. Fletcher. *An Engineering Approach to Digital Design*, page 482. Utah State University, 1980. About metastability.

[8] Sandeep Mandarapu. *Measuring Metastability, Master Project.* Departement of Electrical and Computer Engineering, Southern Illinois University, 2012.

[9] Dictionary of the English Language. *finate state machine.* American Heritage, 2011.

[10] Cambridge Dictionaries Online. *glitch, specialzed electronics.* www.dictionary.cambridge.org/dictionary/english/glitch, 02.11.2015.

9.2. Glossar

Das Glossar dient interessierten Software-Entwicklern, die elektrotechnik-spezifischen Worte zu verstehen.

Asynchrone Signale
Werden Signale in zugewiesen sind sie vorerst ungetaktet, asynchron. Es ist nicht definiert, *wann* exakt das Signal den neuen Wert erhält. Erst wenn ein Signal durch ein Flip-Flop geführt wird, wird es getaktet und seine Signalzuweisung dadurch determinierbar.

Clock Domain
Ein Bereich der Hardware, der mit demselben Takt läuft.

Dekoder
Bezeichnet ein Bauteil, das einen odere mehrere Eingangswert(e) gemäss implementierter Logik in einen Ausganswert wandelt.

Durchlaufverzögerung
Wird englisch *propagation delay* genannt und bezeichnet die Zeit, die Daten vom Eingang bis zum Ausgang des Bauteils brauchen.
Die Durchlaufverzögerung beträgt beim Cylone IV 4 ns [1].

Finate State Machine (fsm)
"A model of a computational system, consisting of a set of states, a set of possible inputs, and a rule

to map each state to another state, or to itself, for any of the possible inputs." [9]
Auf deutsch"Ein Model in Rechensystemen, das aus einem Satz aus Zuständen, möglichen Eingängen und Regeln wie man von einem Zustand zum nächsten, oder zu sich selbst, für alle möglichen Eingänge gelangt. "

Flip-Flops
Grundbaustein der Digitalen Logik. Das Flip-Flop speichert seinen Wert, den es am Eingang erhält am Ausgang.

Glitch
Im technischem Bereich bedeutet *glitch* gemäss Cambridge Dictionaire "a sudden unexpected increase in electrical power, especially one that causes a fault in an electronic system " [10],
auf deutsch "eine plötzliche, unerwartete Spannungserhöhung, die insbesondere ein Fehlverhalten im elektronischen System verursacht".

Hold Time
Ist die minimale Zeit, in der die Inputdaten *nach* der Taktflanke stabil sein müssen.
Die hold-Zeit beträgt beim Cyclone IV E 0 ns [1].

Hot Plug
Bezieht sich auf die Hardware-Umsetzung einer *finate state machine*. Gewöhnlich braucht es für 2^n Zustände n Flip-Flops. Bei Hot Plug braucht es für n Zustände n Flip-Flops, denn jeder neue Zustand wird durch eine '1' am n-ten Flip-Flop detektiert. Alle anderen Flip-Flop-Werte sind auf '0'. Die logische Schaltung für eine *Hot Plug fsm* wird durch den direkten Bezug einer gesetzten '1' zum Zustand einfach.

Kathodenstrahl Oszilloskop, KO
Bezeichnet ein elektronisches Messgerät, das ein Signale analog als Spannungen mit deren zeitlichem Verlauf am Bildschirm ausgibt.

Others
Bezeichnet in einem Swicht-Case in VHDL alle anderen Möglichkeiten, die nicht abgefragt werden. Es dient dem System einen definierten Zustand zu geben, falls etwas Unerwartetes eintrifft.

Pfadzeit
Bezeichnet die Zeit, die ein Signal von einem Flip-Flop zum nächsten braucht.

Setup Time
Minimale Zeit, in der Inputdaten stabil sein müssen be*vor* ein Taktflanke die Daten triggert.
Die setup-Zeit beträgt beim Cyclone IV E 10 ns [1]

State
Bezeichnet den aktuellen Zustan einer finale state machine.

Textbasierte Testbench In VHDL wird die Simulation der Signale in einer Testbench aufgesetzt. In der Testbench werden die Signalanregungen, stimuli, definiert, und die zeitlichen Abläufe unter Signalen. Für eine Testbench ist eine eigene Software notwendig.
Eine textbasierte Testbench liest die stimuli aus einem File ein. Zudem können im File die zu erwartenden Ergebnisse definiert sein.

Token
Bezeichnen Elemente in einer Reihe von strukturierten Daten.

Quartus
IDE von altera zum Kompilieren, Synthesizieren und einbauen von IPs für die altera FPGAs.

A. Offizielle Aufgabenstellung

Beschreibung der Projektarbeit Pa15_gelk_1

In dieser Projektarbeit sollen Versuche entwickelt werden, die für das Modul DTP2 verwendet werden können. Die Arbeit besteht aus zwei Teilen:

Im ersten Teil der Arbeit sollen Versuche entwickelt werden, mit denen folgende Timing Artifakte demonstriert werden können. Dies soll zum zu einem vertieften Verständnis der digitalen Design Grundlagen führen.

- Erzeugung von Glitches mit einem Zähler und nachgeschaltetem Dekoder. Sichtbarmachung der Glitches mit einem Oszilloskop. Betätigen des asynchronen Resets vom Decoder aus.
- Provozieren und sichtbarmachung von Metastabilen Zuständen. Hierfür kann z.B. eine Schaltung mit zwei asynchronen externen Takten aufgebaut werden.

Im zweiten Teil soll mit dem dem Direct Digital Synthesis Verfahren ein Synthesizer mit vielfältigen Klangfarben entwickelt werden. Damit kann anspruchsvolle digitale Schaltungstechnik umgesetzt werden. Zum erreichen der Klangvielfalt können mehrere DDS Generatoren gleichzeitig, mit unterschiedlichen Frequenzen und Phasen betrieben werden. Möglich ist auch eine Frequenzmodulation mit einem zweiten Generator oder Ändern des Volumens mit einer Hüllkurve. Die Ansteuerung soll mit Hilfe eines MIDI Interfaces, welches Polyphonie (mehrere Klaviertasten gleichzeitig gedrückt) unterstützt. Die Implementierung soll im FPGA erfolgen. In der Implementierungsphase der Arbeit soll das Timing der FPGA Implementierung genau betrachtet werden.

Am Ende soll eine Referenzimplementierung in Anlehnung an den Yamaha DX7 für das Modul DTP2 entstehen

B. Aufgabenspezifikation für den zweiten Teil

- Midi Interface for Keyboard für Polyphonie nach Konzept von gelk
  o 10 Frequenz Control Ausgänge zur Steuerung der Tonhöhe des Generators
  o 10 On/Off Ausgänge Ton on/off
  o UART wird geliefert von gelk
  o VHDL wird von Grund auf neu erstellt.
- 10 DDS implementieren und mit Mischer Mischen
- Script basierte Testbench. Testbench erzeugt serielle Midi Dtaen, so wie sie auf dem DIN Stecker vorkommen (logisch)
- Testbench liest eine Testscript Datei ein, in welcher die Tastendrücke eines Keyboards abgebildet werden können. Midi Poliphony Spec muss durch die Testbench unterstützt werden können. Velocity muss nicht unterstützt werden.
- FM Modulation – Tetstbench im Matlab
- Kein VHDL code ohne Testbench.
- Block level testbench. Unit Tests.

Abgrenzung:

- Keine Hüllkurve
- Keine Ausgabe der Velocity aud Midi controller
- Kein Bluetotth

Zeitplan:

- 2.5 Wochen Midi Controller incl. 10 DDS
- 2.5 Wochen FM Synthese

Unterstützung:

- Midi Controller/gelk
- FM-Synthese/rosn

Falls Midi nicht zum geplanten Zeitpunkt fertig wird, wird FM-zurückgestellt. Alle oben genannten Punkte sind Pflicht. Nicht Fertigstellung hat Einfluss auf die Benotung.

C. Konzept Keyboard Dekoder

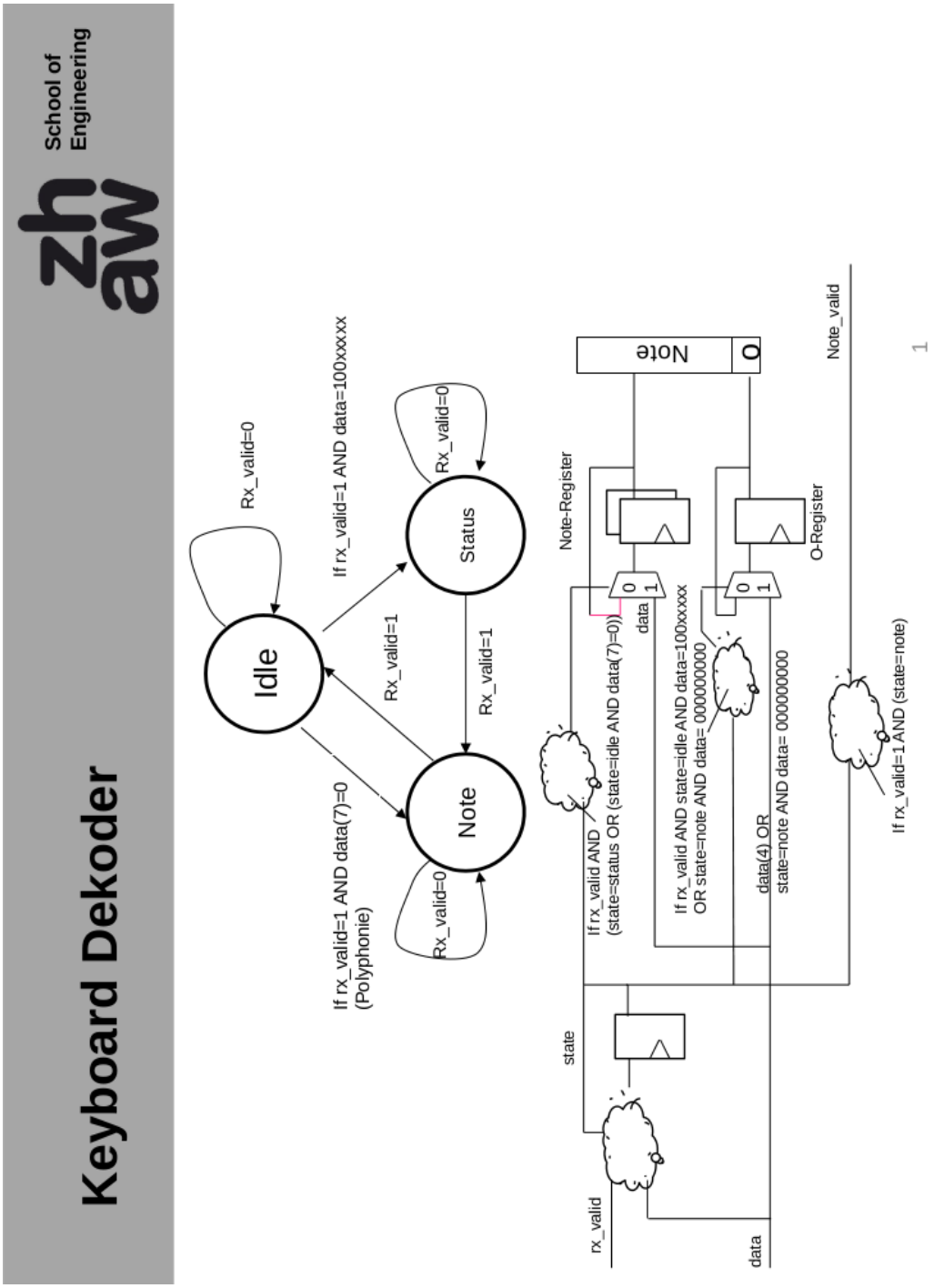

Abbildung C.1.: Midi Controller Spezifikation

D. Konzept Polyphonie

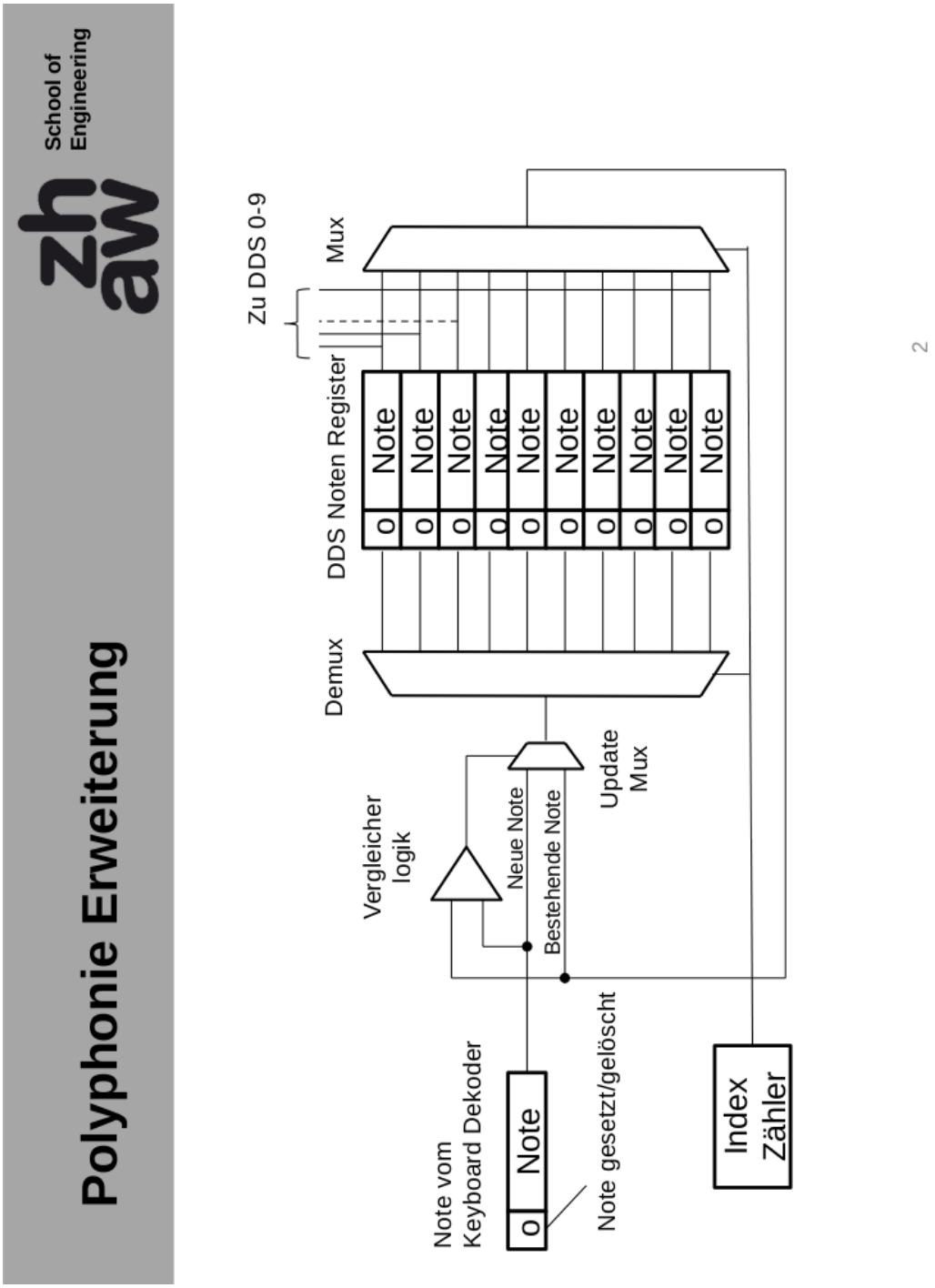

Abbildung D.1.: Spezifikation Polyphonie-Block

E. CD mit Projektdateien

F. Top Synthesizer

In die bestehenden Blöcke und Signale wird das MIDI Interface wiefolgt eingebaut:

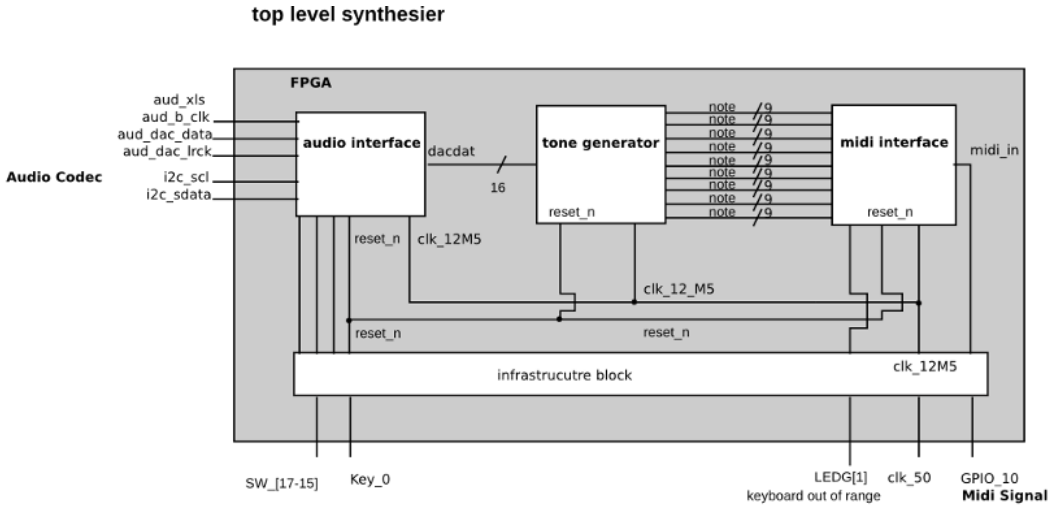

Abbildung F.1.: Top Synthesizer mit MIDI Interface: Blockschaltbild

Hier ist das Konzept der Umsetzung des MIDI Interface detaillierter beschrieben:

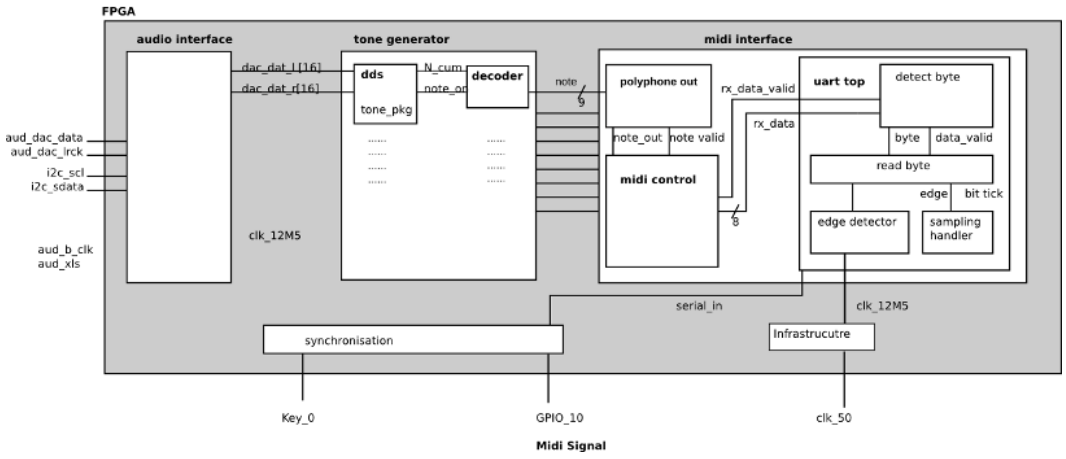

Abbildung F.2.: Top Synthesizer mit MIDI Interface: Detailansicht

G. In- und Outputdatei der textbasierte Testbench

Datei mit Testbefehlen für die Testbench

reset 00 00 00 00 00 00 00 00 00
check 00 00 00 00 00 00 00 00 00
singl 55 90 27 80 27 90 05 00 00
check 00 00 27 00 00 00 05 00 00
singl 55 90 73 80 73 90 16 00 00
check 00 00 00 00 00 00 16 00 00
polyp 71 55 02 55 33 55 08 00 00
check 71 02 33 00 00 00 00 00 03
polyp 71 00 16 55 20 55 33 00 00
check 00 00 16 00 20 00 00 00 04
polyp 02 55 03 00 20 00 40 55 00
check 02 00 16 00 40 00 00 00 03

Das Testergebnis in der Datei

Automatically generated outputfile

Read file with commands in

---

reset
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note number: 00

check
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note:00

Read attribut: 00
Read note number: 00

singl
Read note:55
Read attribut: 90
Read note:27
Read attribut: 80
Read note:27
Read attribut: 90
Read note:05
Read attribut: 00
Read note number: 01

check
Read note:00
Read attribut: 00
Read note:27
Read attribut: 00
Read note:00
Read attribut: 00
Read note:05
Read attribut: 00
Read note number: 01

... etc

polyp
Read note:02
Read attribut: 55
Read note:03
Read attribut: 00
Read note:20
Read attribut: 00
Read note:40
Read attribut: 55
Read note number: 03

check
Read note:02
Read attribut: 00
Read note:16
Read attribut: 00
Read note:40
Read attribut: 00
Read note:00
Read attribut: 00
Read note number: 03

Number of read lines from file: 12
Finished read whole file